ELSEVIER

Artificial Intelligence 74 (1995) 207-248

Artificial Intelligence

Efficient algorithms for qualitative reasoning about time*

Alfonso Gerevini[a], Lenhart Schubert[b]

[a] *IRST—Istituto per la Ricerca Scientifica e Tecnologica, I-38050 Povo, Trento, Italy* *

[b] *Computer Science Department, University of Rochester, Rochester, NY 14627, USA*

Received December 1993; revised March 1994

Abstract

Reasoning about temporal information is an important task in many areas of Artificial Intelligence. In this paper we address the problem of scalability in temporal reasoning by providing a collection of new algorithms for efficiently managing large sets of qualitative temporal relations. We focus on the class of relations forming the Point Algebra (PA-relations) and on a major extension to include binary disjunctions of PA-relations (PA-disjunctions). Such disjunctions add a great deal of expressive power, including the ability to stipulate disjointness of temporal intervals, which is important in planning applications.

Our representation of time is based on *timegraphs*, graphs partitioned into a set of chains on which the search is supported by a metagraph data structure. The approach is an extension of the time representation proposed by Schubert, Taugher and Miller in the context of story comprehension. The algorithms herein enable construction of a timegraph from a given set of PA-relations, querying a timegraph, and efficiently checking the consistency of a timegraph augmented by a set of PA-disjunctions. Experimental results illustrate the efficiency of the proposed approach.

1. Introduction

Representing and reasoning about qualitative temporal information is essential for many tasks of Artificial Intelligence. In several areas, including planning [4,5], plan recognition [21,42], natural language understanding [3,28,34] and diagnosis of technical systems [30], temporal knowledge may take the form of collections of qualitative

* This is a revised and extended version of a paper that appeared in *Proceedings IJCAI-93*, Chambéry, France (1993), and of a paper presented at the *Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR94)*, Bonn, Germany (1994).

* Corresponding author. E-mail: gerevini@irst.it. Phone: +39-461-314-333

SSDI 0004-3702(94)00016-T

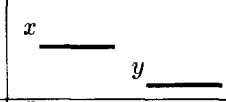
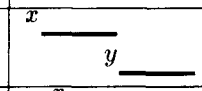
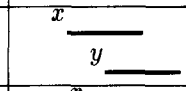
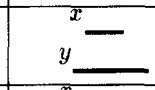
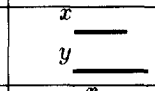
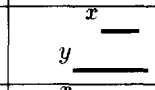
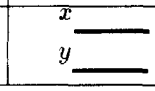
Relation	Meaning
x before y y after x	
x meets y y met-by x	
x overlaps y y overlapped-by x	
x during y y contains x	
x starts y y started-by x	
x finishes y y finished-by x	
x equal y	

Fig. 1. The thirteen basic relations of the Interval Algebra.

relations between time points or intervals. Temporal reasoning tasks include determining consistency (satisfiability) of such collections, finding a consistent scenario (an interpretation for all the temporal variables involved) and deducing new relations from those that are known (or computing their closure).

For example, Allen and Koomen [5] introduced a planning system which is based on such forms of temporal reasoning and which uses an Interval Algebra (IA) formed from subsets of thirteen basic relations (see Fig. 1) [2]. Their framework allows different modes of plan reasoning such as plan construction [4,5] and plan recognition [21,42] to be formalized. By allowing for temporal constraints among properties of domain objects and actions available to a planner, they are able to reason effectively about simultaneous actions, and to construct plans which take into account interactions among contemplated actions. The task of checking the consistency of these relations is essential in the planning algorithm presented by Allen in [4]. Unfortunately, since the temporal relations used belong to IA, and reasoning with such an algebra is NP-complete [40,41], this task can be the bottleneck in the performance of the planning algorithm on large-scale problems.

In [4] Allen indicates two possible ways to avoid spending exponential time in temporal reasoning. One is to reduce the class of temporal relations to a tractable subset such as the *pointizable interval relations* [24,39]. (This set of relations, called SIA in [37], consists of those relations in IA that can be translated into conjunctions of point relations between endpoints of intervals.) The other is to use an incomplete temporal reasoner that works in polynomial time but which can fail to detect inconsistencies

among temporal constraints. As pointed out by Allen, the limitation of the first method is that some practically essential relations such as disjointness relations cannot be expressed. Disjointness is needed, for example, in constraining two actions that require dedicated use of the same resources (agents, tools, pathways, etc.) to be nonoverlapping in time. The limitation of the second is that it may allow the construction of plans which are temporally inconsistent (and hence the planning algorithm is in general not sound). Moreover, in both cases traditional methods for temporal reasoning based on constraint propagation can incur prohibitive costs for large databases.

Since Allen's work on binary interval relations, numerous researchers have further investigated temporal reasoning based on constraint propagation techniques [11,25–27,38,39,41]. However scalability has remained a problem. Algorithms exist for SIA running in $O(m)$ space and $O(n^2)$ time for finding a scenario [37], and in $O(n^2)$ space and $O(n^4)$ time for computing the closure [39] (for m relations and n intervals). Nebel and Bürckert have identified the largest tractable subalgebra of IA containing the 13 basic IA-relations, which they term the ORD-Horn subalgebra [29]. The set of relations in the ORD-Horn subalgebra strictly contains the relations of SIA but does not include the disjointness relations of IA. ORD-Horn consistency testing can be accomplished in $O(n^2)$ space and $O(n^3)$ time, and computing the closure in $O(n^2)$ space and $O(n^5)$ time. Unfortunately these bounds are still unacceptable for domains in which a large database of relations needs to be managed [1].

Recently, other approaches based on graph algorithms have been proposed whose main characteristic is that of providing better performance in practice compared to the more traditional constraint-based approaches [9,12,14,15,17–19,28]. The present paper follows a similar direction. Our goal is to efficiently manage large data sets of qualitative temporal relations including at least the pointizable relations and disjointness relations, without sacrificing completeness. Thus we begin with the Point Algebra (PA) [41] and add 4-point relations expressed as binary disjunctions of PA-relations. The elements of PA are the relations $\{<, >, =, \leqslant, \geqslant, \neq, \leqslant>, \emptyset\}$, through which all the relations of SIA can be represented [24]. Disjunctions of PA-relations allow the representation of interval disjointness relations and cover a larger class of IA-relations than SIA and the ORD-Horn subalgebra.[1] Moreover, they can express some non-binary interval relations such as *interval I before interval J or after interval K* which are not in IA.

Our approach is based on ideas derived from a temporal reasoning system developed in the context of natural language comprehension [28,31,32]. In this system temporal relations are represented through graphs, called *timegraphs*, whose vertices represent points and whose edges represent temporal relations. The main characteristics of timegraphs are their partitioning into a set of chains, which are sets of linearly ordered points, and their use of a *metagraph* structure to guide the search processes across the chains. One advantage of this approach is that the space complexity can be linear in the number of stipulated relations. The other advantage is the efficiency in domains such as planning [1] and story understanding [28] in which the temporal data tend to fall naturally into chain-like aggregates. Essentially this is because the "worlds" described

[1] Each of the relations in the ORD-Horn subclass of IA can be translated into a collection of PA-relations and (binary) PA-disjunctions with at most one disjunct in $\{\leqslant, =\}$ [29].

in plans and stories consist of individuals (or sets of related individuals) each moving through a course of actions and events, creating a trajectory in time. Building timegraphs in practice takes much less time than computing closure (the *minimal network* in constraint propagation terminology), and the amount of time spent in querying relations is nearly constant.

In this paper we present new efficient algorithms for building timegraphs from sets of PA-relations and for checking the consistency of PA-disjunctions. Among them there is an algorithm for dealing efficiently with "not equal" relations ($\neq$) which were not considered in previous timegraph algorithms. To our knowledge, the only authors who have proposed algorithms for reasoning efficiently with $\neq$ relations in the context of the Point Algebra are Ghallab and Mounir Alaoui [18] and van Beek and Cohen [39]. Neither has treated the problem quite satisfactorily. In fact, the correctness of the algorithm proposed by van Beek [37,38] is based on a lemma whose proof as given in [39] turns out to be incorrect. However, in [16] we provide a new proof for the lemma which shows that van Beek's algorithm is indeed correct. In [18] the $\neq$ relations are only partially treated as the proposed algorithms cannot derive some strict orderings induced by $\neq$ relations.

Section 2 introduces our framework formally. Section 3 shows the algorithms for building a timegraph and for querying relations. In Section 4 we deal with the problem of efficiently checking the consistency of a timegraph augmented with a set of disjunctions of PA-relations. We present a general method applicable to such augmented timegraphs, regardless of the kinds of disjunctions involved, and provide a specialized algorithm for binary disjunctions of strict inequalities. The method is based on two main steps: the first exploits the information provided by the timegraph to preprocess the initial set of disjunctions, reducing it to a logically equivalent subset; the second performs a search to check the consistency of the reduced set which uses a form of selective backtracking [7,33] and a "forward propagation" technique to greatly enhance efficiency. The preprocessing phase is worst-case polynomial, and in principle is strong enough to subsume consistency checking for the ORD-Horn class of interval relations.

Section 5 reports experimental results from *TimeGraph-II* (TG-II), a system in which the algorithms described in the previous sections have been implemented. Section 6 gives conclusions and indicates future work.

2. Representing temporal relations through graphs

In this section we introduce the definitions and theorems on which the proposed approach is based.

Definition 2.1. A *temporally labeled graph* (*TL-graph*) is a graph with at least one vertex and a set of labeled edges, where each edge (v, l, w) connects a pair of distinct vertices v, w. The edges are either directed and labeled $\leqslant$ or $<$, or undirected and labeled $\neq$.

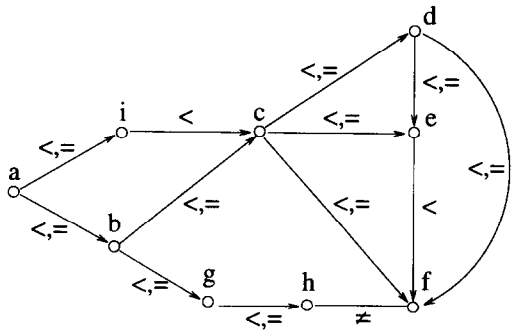


Fig. 2. An example of *TL*-graph.

Fig. 2 shows an example of a *TL*-graph. We assume that every vertex of a *TL*-graph has at least one name (time-point variable) attached to it. If a vertex has more than one name, then they are alternative names for the same time point. The name sets of any two vertices are required to be disjoint. More formally, writing P for the set of time-point variables attached to a *TL*-graph and V for its set of vertices, we assume that there exists a surjective function PV from P to V.

A *model* of a *TL*-graph is an interpretation of the vertex names as elements of a totally ordered set T (with strict ordering $<$), such that all names attached to the same vertex denote the same element and the interpretations of names attached to distinct vertices satisfy the constraints expressed by the edge(s), if any, connecting those vertices. The notions of an interpretation and of a model for a *TL*-graph can be formally specified as follows:

Definition 2.2. Given a *TL*-graph G, an *interpretation* of G is a triple $\langle T, I, R\rangle$, where T is a totally ordered set (with ordering $<$); I is a function $I : P \rightarrow T$ such that for all $p_1, p_2 \in$ P *if* $PV(p_1) = PV(p_2)$ *then* $I(p_1) = I(p_2)$; and R is a function mapping each label l on the edges of G ("$<$","$\leqslant$" or "$\neq$") into the corresponding binary relation $R(l)$ on T ($<$, $\leqslant$ or $\neq$).[2]

Definition 2.3. Given a *TL*-graph G, a *model* of G is an interpretation such that if (v_1, l, v_2) is an edge of G, then for all $p_i, p_j \in P$ satisfying $PV(p_i) = v_1$ and $PV(p_j) = v_2$, $\langle I(p_i), I(p_j)\rangle \in R(l)$.

Definition 2.4. A *TL*-graph is *consistent* if and only if it has at least one model.

Definition 2.5. Two or more *TL*-graphs are *logically equivalent* if and only if they have the same models.

A path of length n from v_0 to v_n is a sequence of n triples $(v_0, l_1, v_1), \ldots, (v_{n-1}, l_n, v_n)$ where v_i $(0 \leqslant i \leqslant n)$ are vertices and l_j $(1 \leqslant j \leqslant n)$ are labels (relations) on edges.

[2] For instance if l = "$<$" then $R(<)$ is the $<$-relation, i.e., the set of pairs $\langle t_1, t_2\rangle$ such that $t_1 < t_2$ and $t_1, t_2 \in T$. (Analogously for "$\leqslant$" and for "$\neq$".)

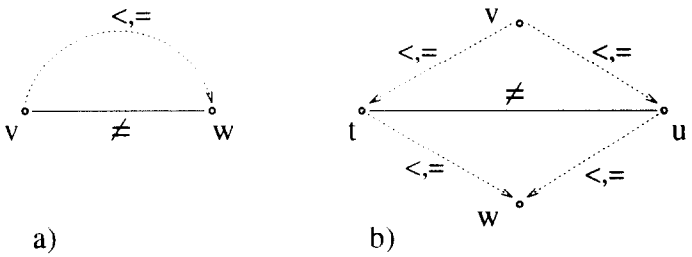


Fig. 3. The two kinds of implicit $<$ relation. Dotted arrows indicate paths, solid lines $\neq$-edges. In both the graphs there is an implicit $<$ relation between v and w.

Definition 2.6. In a *TL*-graph we call a path a $\leqslant$-*path* if each label l_i is $\leqslant$ or $<$. A $\leqslant$-path is a $<$-*path* if at least one of these labels is $<$.

Definition 2.7. A $\leqslant$-path ($<$-path) of length n from v_0 to v_n, $n \geqslant 1$, is a $\leqslant$-*cycle* ($<$-*cycle*) if $v_0 = v_n$. A *TL*-graph is *acyclic* if it does not contain any $\leqslant$-cycle.

Theorem 2.8. *A TL-graph is consistent* iff *it does not contain any $<$-cycle, or any $\leqslant$-cycle that has two vertices connected by an edge with label $\neq$.*

Proof (Sketch). The "only if" direction is obvious from irreflexivity of $<$, and from equality of vertices on a $\leqslant$-cycle. The "if" direction is proved by induction on the number of $\neq$-edges. When there are none, the absence of $<$-cycles guarantees consistency. In the induction step, we note that we can consistently add $(v, \neq, w)$ just in case we can consistently add $(v, <, w)$ or consistently add $(w, <, v)$. We use the former if there is a $\leqslant$-path from v to w, and the latter otherwise, obtaining a situation covered by the induction assumption. □

Definition 2.9. A *TL*-graph contains an *implicit* $<$ relation $v < w$ if there is no $<$-path from v to w and either there is an edge between v and w with label $\neq$ and a $\leqslant$-path (but no $<$-path) from v to w (see Fig. 3(a)); or there exist two vertices t and u such that there is an edge between t and u with label $\neq$ and $\leqslant$-paths (but no $<$-path) from v to u, u to w, v to t, and t to w (see Fig. 3(b)). The graphs isomorphic to the one given in Fig. 3(b) are called $\neq$-*diamonds*.

Definition 2.10. An *explicit* graph for a given *TL*-graph G is an acyclic *TL*-graph logically equivalent to G and with no implicit $<$ relations.

Theorem 2.11. *An explicit TL-graph entails $v = w$* iff *v and w are alternative names of the same vertex; it entails $v \leqslant w$* iff *there is a $\leqslant$-path from v to w; it entails $v < w$* iff *there is a $<$-path from v to w, and it entails $v \neq w$* iff *there is a $<$-path from v to w or from w to v, or there is an edge $(v, \neq, w)$.*

Proof (Sketch). Since by definition an explicit *TL*-graph is acyclic, it is easy to show that only equalities between point-variables in the same set of names of a vertex can be entailed. Regarding the entailment of the other relations, the claim is still easy to establish when there are no $\neq$-edges in the graph. When there are $\neq$-edges, the idea is to show that if there is no $\leqslant$-path ($<$-path) from v to w, we can "decide" any $\neq$-edge (making it a $<$-edge in one direction or the other) without creating cycles, without creating a $\leqslant$-path ($<$-path) from v to w, and without forfeiting the "explicit graph" property. Thus all the $\neq$-edges can be decided, and we end up with a consistent graph that entails the original one, yet doesn't entail $v \leqslant w$ ($v < w$). The reason we can avoid creating a $\leqslant$-path ($<$-path) from v to w is that if both ways of deciding a $\neq$-edge gave such a path, then a $\neq$-diamond would have to be present (formed by v, w, and the vertices connected by the $\neq$-edge), contrary to the "explicit graph" property. And the reason we do not forfeit the "explicit graph" property is that a $<$-edge replacing a $\neq$-edge cannot create paths that support an *implicit* $<$ relation (they can only support an explicit $<$ relation). □

Remark 2.12. This theorem has two important consequences. One is that it enables us to write linear time procedures which obtain the strongest relation between two points entailed by an explicit graph (called the *minimal label* by van Beek and Cohen [39]). Secondly, it provides the basis for a new proof of their Lemma 1 in [39] stating that any path-consistent nonminimal network of relations[3] taken from PA must include a four-vertex subgraph isomorphic to the graph in Fig. 3(b). In [16] we present such a proof.

The value of this new proof lies in the fact that the one given by van Beek and Cohen is not correct. This becomes evident if one observes that the four-vertex constraint network obtained by replacing the $\leqslant$-paths in Fig. 3(b) with $\leqslant$ edges and adding the edge $(v, \{<, =\}, w)$ is not the only four-vertex *path consistent network* that, up to isomorphism, can be derived in the last step of the proof of van Beek and Cohen. In fact, the network obtained by replacing $(v, \{<, =\}, u)$ with $(v, \{<, =, >\}, u)$ and $(u, \{<, =\}, w)$ with $(u, \{<, =, >\}, w)$ is another possible network. This contradicts what van Beek and Cohen assert at the end of their proof.

We will show how linear time procedures for querying relations can be designed in Section 3.5.

3. The construction of a timegraph

Given a set S of binary relations in the Point Algebra we first build a *TL*-graph whose vertices correspond to the variables (time points) in S, and whose edges correspond to

[3] A network is minimal if there is exactly one labeled edge for each pair of distinct vertices and all the labels are minimal. A *path-consistent* network is one in which for any three vertices u, v, w, any of the possible relations $\{<, >, =\}$ allowed by the label of the edge joining u and v is consistent with the labels joining u and w, and v and w. This lemma is particularly important in the constraint propagation approach to temporal reasoning because it allows computation of the *minimal network* for a set of relations in the point algebra by an algorithm taking $O(n^4)$ time (for n vertices) [39].

the relations (note that = relations are translated by creating only one vertex for each set of explicitly equated variables, and labeling that vertex with that set of variables). For example, the relation $x \leqslant y$ is translated into a pair of vertices v_x, v_y and a directed edge from v_x to v_y with label $\leqslant$.

In the following we will assume that the graph is represented by a double adjacency list which stores for each vertex the list of predecessors and the list of successors.

From the *TL*-graph we then form a *timegraph* whose definition is:

Definition 3.1. A *timegraph* is an acyclic *TL*-graph partitioned into a set of *time chains*, such that each vertex is on one and only one time chain. A time chain is a $\leqslant$-path, plus possibly *transitive edges* connecting pairs of vertices on the $\leqslant$-path.

Distinct chains of a timegraph can be connected by *cross-edges* (these, and certain auxiliary edges, will also be called *meta-edges*). Vertices connected by cross-edges are called *cross-connected vertices* (or *metavertices*).[4]

The construction of a timegraph from a *TL*-graph consists of four main steps: consistency checking, ranking of the graph, formation of the chains and making all implicit $<$ relations explicit.

3.1. Checking consistency

Determining the consistency of a *TL*-graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, can be accomplished in two steps. The first step consists of the identification of all the strongly connected components (SCC) [8] of the *TL*-graph derived from G ignoring the $\neq$ relations. The second step consists of checking if any of the SCCs contains a pair of vertices connected by an edge with label $<$, or $\neq$. It can be shown from Theorem 2.8 that a *TL*-graph is consistent if and only if such a SCC does not exist. When the graph is consistent, each SCC can be collapsed into any arbitrary vertex v within that component. All the cross-component edges entering or leaving the component are transferred to v, and so are all names attached to vertices within the component. The edges within the component can be eliminated. It is clear from the definition of a SCC that the resulting graph is acyclic and that this resultant graph is logically equivalent to the original one. It is well known that the computation of the SCCs can be accomplished in time $O(n + e)$ (where n is the number of vertices and e the number of edges) using Tarjan's algorithm [36]. Checking the existence of $<$, $\neq$ edges between vertices in a SCC and collapsing each SCC into a single vertex are linear tasks in the number of the edges. It follows that the global time complexity is $O(e)$. This provides a proof of the next theorem (a similar theorem is also given by van Beek [37]):

Theorem 3.2. *A TL-graph can be recognized as being inconsistent, or if it is consistent, collapsed into a logically equivalent acyclic TL-graph in* $O(e)$ *time, where e is the number of edges.*

[4] This is a slight departure from the terminology of [28,32].

```
INPUT: A consistent TL-graph G = (V, E), an integer k
OUTPUT: The ranks of the vertices in G
1.    topologically sort the vertices in G using only <, ≤-edges;
2.    for each vertex v ∈ V do rank(v) := −∞;
3.    rank(universal-start-time) := 0;
4.    for each vertex u taken in topologically sorted order
5.       do for each vertex v such that (u, {<, ≤}, v} ∈ E do
6.             if rank(v) < rank(u) + k then
7.                rank(v) := rank(u) + k.
```

Fig. 4. An algorithm for computing the vertex ranks.

3.2. Ranking the graph

Given a *TL*-graph G, we define the *universal start time* of G as a special vertex with no predecessors and whose successors are the set of vertices in G which have no other predecessors. Each edge leaving from the universal start time has label $\leqslant$.

In accordance with the definition given by Ghallab and Mounir Alaoui in [18], the *rank* of a vertex v is defined as the length of the longest $\leqslant$-paths from the universal start time to v, times the distance increment k. (I.e., rank increases in steps of size k along maximal-length paths.) In our terminology the rank of a vertex is also called the *pseudotime* of that vertex. The main purpose of the ranks is bounding the search at query time. In fact, using the ranks, it is often possible to obtain the strongest relation between two vertices that is entailed by the graph in constant time, i.e. without performing any search. We will show how this is possible in the next section, while in Section 3.5 we discuss the query algorithms further. Fig. 4 shows an algorithm for computing the ranks for an acyclic *TL*-graph based on the DAG-single-source-longest-paths algorithm reported in [8] and taking $O(n + e)$ time.

3.3. Forming the time chains and the metagraph

The main computational importance of time chains is that, given a pair of vertices belonging to the same chain, it is possible to compute the strongest relation entailed by the graph in constant time, whereas in general this task requires a graph search linear in the number of edges (unless of course we precompute all $O(n^2)$ minimal labels). The constant time algorithm is given in Fig. 5. This result is achieved by using the pseudotimes of four vertices and an additional, possibly null, link for each vertex. Following [28], we call this link the *nextgreater* link, defined in the following way:

Definition 3.3. Given a vertex v, the *nextgreater of* v (written nextgreater(v)) is the nearest successor v' of v that is on the chain of v such that $v < v'$ is entailed by the graph. If v' does not exist, then nextgreater(v) is null.

We describe how the nextgreaters can be efficiently computed in the next section. Since vertices on the same chain support constant time queries, it is desirable to keep the number of time chains to a minimum in building the timegraph. Another desirable constraint is that the number of cross-edges be minimal.

```
INPUT: two vertices v1 and v2 on the same chain.
OUTPUT: the strongest relation between v1 and v2.
1.    if pseudotime(v1) < pseudotime(v2) then
2.      if pseudotime(nextgreater(v1)) ⩽ pseudotime(v2) then
3.        return v1 < v2 else return v1 ⩽ v2
4.    else if pseudotime(nextgreater(v2)) ⩽ pseudotime(v1)
5.          then return v2 < v1 else return v2 ⩽ v1.
```

Fig. 5. Algorithm for determining the relation between vertices on the same chain of a timegraph.

```
INPUT: a consistent TL-graph G = (V, E)
OUTPUT: G with vertices assigned to chains (using chain indices c = 1, 2, ...).
1.    X := list of vertices in V; c := 1;
2.    for each vertex v ∈ X chain(v) := nil;
3.    w := a vertex in X with the highest rank;
4.    chain(w) := c; remove w from X;
5.    v := a vertex in V such that chain(v) = nil and (v, l, w) ∈ E, where l ∈ {<, ⩽}
          (preferring a vertex connected to w by a <-edge), or nil if there is no
          such vertex;
6.    if v ≠ nil then
        begin
7.      w := v; goto 4
        end
8.    else if X is not empty then
        begin
9.      c := c + 1; goto 3
        end.
```

Fig. 6. Algorithm for creating chains.

Definition 3.4. A timegraph is an *optimal timegraph* if it has the least possible number of time chains and the least possible number of cross-edges.

The problem of minimizing the number of chains could be solved in time $O(n^3)$ using a Maximum Flow algorithm [8]. However, this complexity is too high for our purposes; for scalable temporal reasoning, the preprocessing complexity of the data should be linear, or not much worse than linear, in the size of the data.

Fig. 6 shows an algorithm for creating a set of time chains from a given *TL*-graph that attempts to minimize the number of cross-edges and that works well in practice. It can be shown, using the adjacency list of the preceding vertices, that the time complexity of the algorithm is $O(n+e)$. Fig. 7 shows three time chains formed by using the algorithm on the graph of Fig. 2.

3.3.1. Computing the nextgreater links

The algorithm for computing the nextgreater links consists of two main steps. In the first step the nextgreater links for each vertex v are computed considering only edges connecting vertices on the same chain as v. In the second step these nextgreater links are refined by looking for cross-chain $<$-paths.

Fig. 8 shows an algorithm for performing the first step for a single chain C that takes $O(n_C + e_C)$ time, where n_C is the number of vertices on C and e_C is the number of

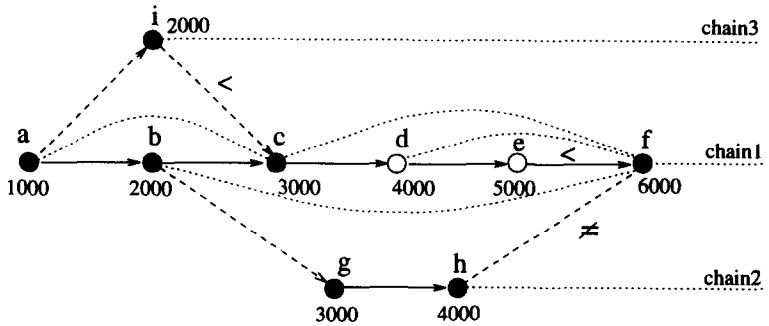


Fig. 7. Timegraph of the *TL*-graph of Fig. 2, with transitive edges and auxiliary edges omitted. The chain edges (in solid lines) and cross-edges (in broken lines) with no label are assumed to be labeled $\leqslant$. The dotted edges represent nextgreater links.

```
INPUT: a time chain C
OUTPUT: the local nextgreater links for C
1.    i := first vertex on C ; n := last vertex on C;
2.    while rank(i) <rank(n) do
3.       k := successor of i on C; strict := false;
4.       while strict = false and rank(k) ⩽ rank(n) do
5.          S = {(j,l,k) | j ∈ C, l ∈ {<, ≠}, and rank(i) ⩽ rank(j) < rank(k)};
6.          if S is not empty then
                begin
7.              r := the vertex with the highest rank in {start(e) | e ∈ S};
8.              strict := true;
9.              for each vertex t on C between i and r do
10.                 nextgreater(t) := k;
11.             i := successor of r on C
                end
12.         else k := successor of k on C
         end{while}
      end.{while}
```

Fig. 8. Algorithm for computing local nextgreaters.

edges between those vertices. The function $start((v_1, l, v_2))$ indicates the vertex between v_1 and v_2 that has the minimum rank. Since two vertices on the same chain may also be connected via cross chain paths, it is clear that the first step is not sufficient for computing nextgreaters. The second step completes this task by performing a search for the $<$-paths that go from one cross-connected vertex of a chain to another on the same chain, starting with an out-going cross-edge and ending with an incoming cross-edge. For example, in Fig. 7 the nextgreater of a is c, but before refinement it was f. Edges with no label have been assumed to be labeled $\leqslant$. The dotted edges represent the nextgreater links. The number labeling each vertex is the pseudotime obtained using increments of 1000.

The second step is in general the most time-consuming in the computation of nextgreaters; in fact since each search can cost $O(e)$ time, the time complexity is $O(n \cdot e)$. However, for timegraphs it is possible to improve this bound significantly with

a negligible cost in space complexity (a constant factor). In order to do that, we need two new links for each cross-connected vertex v. The first points to the first successor of v on the same chain which has outgoing cross-edges, and the second points to the first successor of v on the same chain which has incoming cross-edges. We indicate the vertices pointed to by these two links with *nextout(v)* and *nextin(v)*. Based on the metavertices and the meta-edges of the timegraph, together with these new links, we define the *metagraph* of a given timegraph T:

Definition 3.5. Given a timegraph T, the *metagraph* of T is the graph $\hat{G} = (\hat{V}, \hat{E})$ in which $\hat{V} = \{v \mid v \text{ is a metavertex in } T\}$ and $\hat{E} = \{(v, l, w) \mid (v, l, w) \text{ is a cross-edge in } T\} \cup \{(v_i, nextout(v_i)), (v_i, nextin(v_i)) \mid v_i \in \hat{V}\}$.

It can be shown that $\hat{G}$ can be computed from T in $O(n+e)$ time. The algorithm for refining the nextgreaters on a chain C, whose pseudocode is reported in Appendix A, starts by considering the last vertex on C, v_{start}, that might be refined (i.e., such that it has an outgoing cross-edge and its nextin points to a vertex on C preceding the current nextgreater). Once we have found all the paths from v_{start} to other cross-connected vertices in C, and we have updated the value of nextgreater(v_{start}) accordingly, we "move back" to vertices on C preceding v_{start} and update each of their nextgreater links, until we get to another cross-connected vertex with outgoing cross-edges. Then we initiate another search. Since this search can be performed using the metagraph, the time complexity of the algorithm on chain C is $O(n_C + \hat{n}_C \hat{e})$ time, where n_C is the number of vertices on C and $\hat{n}_C$ is the number of metavertices on C. The global complexity of the algorithm for computing the nextgreaters is then

$$O(e + \sum_C (n_C + \hat{n}_C \cdot \hat{e})) = O(e + n + \hat{n} \cdot \hat{e}).$$

Moreover, the search required for refining the nextgreater of a vertex v can be pruned whenever a vertex with a rank greater or equal to the rank of the current nextgreater of v is reached (by the definition of rank and by the way the chains are formed, it is not possible to have a path from a vertex v to a vertex with a rank equal or greater than the rank of v).

3.4. Dealing with "not equal" relations

Reducing a *TL*-graph (i.e. making explicit all the implicit $<$ relations by the addition of new edges with label $<$) can be the most expensive task in the construction of a timegraph. This is the same as in van Beek's approach whose algorithm [37,38] takes $O(\max(n^2 \cdot e_{\neq}, n^3))$ time, with n the number of time points, and $e_{\neq}$ the number of $\neq$ relations. Since our approach is aimed at managing large data sets, this step is the crucial one. Fortunately, the data structures provided by a timegraph allow this task to be accomplished more efficiently. Fig. 3 shows the two cases of implicit $<$ relations. An algorithm for the first case (Fig. 3(a)) is given in Fig. 9. The time complexity of this algorithm is $O(\hat{e}_{\neq} \cdot (\hat{e} + \hat{n}))$, where $\hat{e}_{\neq}$ is the number of $\neq$ cross-edges in the

INPUT: a consistent *TL*-graph $G = (V, E)$
OUTPUT: G without the implicit $<$ of Figure 3(a)
1. **for each** edge $(v, \neq, w)$ in G
2. **if** there exists a $<$-path from v to w **then**
3. remove $(v, \neq, w)$ from G
4. **else if** each path from v to w is a $\leqslant$-path but not a $<$-path **then**
5. remove $(v, \neq, w)$ from G and add $(v, <, w)$ to G.

Fig. 9. Algorithm for reducing $\neq$ relations.

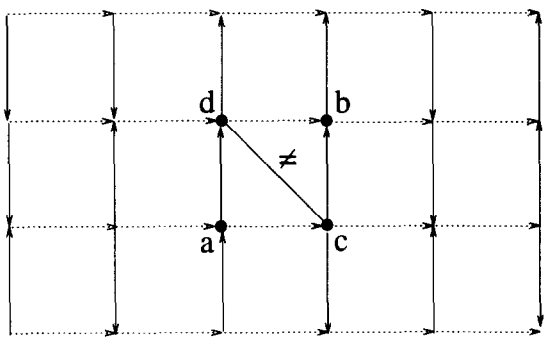


Fig. 10. An example of timegraph with several $\neq$-diamonds of which only the one consisting of the vertices {a,b,c,d} is the smallest. A dotted arrow represents a $\leqslant$-path on a single chain that is not a $<$-path. Solid edges are assumed labeled $\leqslant$.

timegraph,[5] $\hat{e}$ is the number of meta-edges. In fact, using the metagraph, the tests in steps 2 and 3 can be performed in $O(\hat{e} + \hat{n})$ time (see Section 3.5).

In order to make implicit $<$ relations of the second kind (Fig. 3(b)) explicit, a number of $\neq$-diamonds of the order of $e_{\neq} \cdot n^2$ may need to be identified in the worst case. However, for timegraphs only a subset of these needs to be considered. In fact it is possible to limit the search to the *smallest* $\neq$-diamonds, i.e., the set of diamonds obtained by considering for each edge $(v, \neq, w)$ only the *nearest common descendants* of v and w ($NCD(v, w)$) and their *nearest common ancestors* ($NCA(v, w)$). This is a consequence of the fact that, once we have inserted a $<$ edge from a vertex in $NCA(v, w)$ to a vertex in $NCD(v, w)$, we will have explicit $<$-paths for all pairs of "diamond-connected" vertices.

Fig. 10 shows an example of a timegraph with several $\neq$-diamonds of which only one is the smallest and needs to be considered. From the previous observation we can derive a criterion for pruning the search that in practice can give significant time savings. In fact the total number of diamonds that has to be considered is bounded by

$$\sum_{(v, \neq, w)} |NCA(v, w)| \cdot |NCD(v, w)|.$$

Fig. 11 shows an efficient algorithm, based on the computation of the nearest common descendants and ancestors, for making the implicit $<$ relation of $\neq$-diamonds explicit.

[5] Note that if $(v, \neq, w)$ is a transitive edge on a chain we have an implicit $<$ relation of the first kind, and hence it is already considered by the algorithm of Fig. 9.

```
NCD-NCA algorithm
INPUT: a ranked TL-graph G
OUTPUT: the explicit graph of G
1.    for each edge (u, ≠, v) do
          begin
2.        Y := NCD(u,v);
3.        X := NCA(u,v);
4.        for each pair x, y such that x ∈ X and y ∈ Y do
5.            if there is no <-path form x to w then
6.                add the edge (x, <, y) to G
          end

NCD Algorithm
INPUT: a ranked TL-graph G and a pair of vertices u, v
OUTPUT: the nearest common descendants of u and v
1.    for each vertex v ∈ G code(v) := 0; NCD := nil;
2.    if rank(u) ⩽ rank(v) then
3.        OPEN := ((u 1)(v 2))
4.    else OPEN := ((v 2)(u 1));
5.    if there is no item in OPEN with code in {1,2,3} then
6.        return NCD;
7.    w := pop(OPEN);
8.    if code(w)=3 then
9.        add w to NCD and let code(w) := 4;
10.   X := {x | (w, ⩽, x) is an edge of G };
11.   for each x ∈ X do
12.       if x ∈ OPEN then
13.           change code(x) on OPEN to COMBINE[code(w),code(x)]
14.       else add the item (x code(w)) to OPEN in rank order,
              maintaining vertices with the lowest ranks in initial position;
15.   if OPEN = nil then return NCD else goto 5.
```

Fig. 11. Algorithm for reducing ≠-diamonds. The meanings of code(x) are: 1 descendant of u; 2 descendant of v; 3 common descendant of u and v; 4 vertex at which the search can be stopped.

The algorithm uses the list OPEN as a supplemental data structure. Each item in OPEN is a pair (v, q) where v is a vertex to be visited and q is an integer between 1 and 4 called the code of v. The code of a vertex is used to recognize when the vertex is a nearest common descendant and, given the set of vertices already visited, when it is not possible to reach any other nearest common descendants in addition to the ones already reached. The codes of two vertices are combined using a table (COMBINE) that is given in Table 1.

The time complexity of *NCD* (*NCA*) is $O(e)$, but exploiting the timegraph's data structures, we can obtain better performance by limiting the search to the metagraph. This is because we need to consider only ≠ cross-edges (see footnote 5), and if $(v, \neq, w)$ is such a cross-edge, $NCD(v, w)$ and $NCA(v, w)$ must all be metavertices. So, writing $\overline{NCA}(v, w)$ and $\overline{NCD}(v, w)$ respectively for the subset of $NCA(v, w)$ and the subset of $NCD(v, w)$ consisting only of metavertices, the number of ≠-diamonds to be considered is reduced to

Table 1
COMBINE[a]

$c_1 \backslash c_2$	1	2	3	4
1	1	3	3	4
2	3	2	3	4
3	3	3	4	4
4	4	4	4	4

[a] Table for combining the codes of two vertices in the algorithm for reducing $\neq$-diamonds (Fig. 11).

$$\sum_{(v,\neq,w)\in\hat{E}} |\overline{NCA}(v,w)| \cdot |\overline{NCD}(v,w)|.$$

The algorithm for computing $NCA(v,w)$ is analogous to the NCD algorithm.

To conclude the treatment of the $\neq$ relations, we observe that the addition of new $<$ edges to the timegraph requires, in general, the updating of the nextgreaters and the nextout, nextin links.

3.5. Query algorithms

Given an explicit *TL*-graph, Theorem 2.11 permits us to derive the strongest relation between two time points that is entailed by the graph, just by looking for all the paths connecting the two corresponding vertices. Furthermore, in timegraphs there are four special cases in which those paths can be obtained in constant time. Given two points p_1, p_2 the first case is the one where p_1 and p_2 are alternative names of the same point. The second case is the one where the vertices v_1 and v_2 corresponding to p_1, p_2 are on the same chain (see Section 3.3). The third case is the one where v_1 and v_2 are not on the same chain and have the same rank, and there is no $\neq$ edge between them (the entailed relation is $\{<,=,>\}$). The fourth case is the one in which there is a $\neq$ edge joining v_1 and v_2. Provided that during the step of making implicit $<$ relations explicit we remove all redundant $\neq$ edges (i.e., $\neq$ edges between vertices connected by a $\leqslant$-path), $p_1 \neq p_2$ is the strongest relation entailed by the graph just in case there exists a cross-edge with label $\neq$ connecting the corresponding vertices v_1, v_2.

In the remaining cases an explicit search of the graph needs to be performed. If there exists at least one $<$-path from v_1 to v_2, then the answer is $v_1 < v_2$. If there are only $\leqslant$-paths (but no $<$-paths) from v_1 to v_2, then the answer is $v_1 \leqslant v_2$. Analogously for paths from v_2 to v_1. An algorithm for accomplishing this task can be derived by a slight adaptation of the single-source-longest-paths algorithm for directed acyclic graphs reported in [8]. This algorithm has a time complexity of $O(n+e)$ but, exploiting again the timegraph's data structures, and in particular the metagraph, we can adapt it to obtain a complexity of $O(k+\hat{e}+\hat{n})$, where k is the constant corresponding to the time required by the four special cases. Moreover, as in the computation of the nextgreater links in Section 3.3.1, the search from a vertex v_1 to a vertex v_2 can be pruned whenever a vertex with a rank greater then or equal to the rank of v_2 is reached.

I not-before J	$\Leftrightarrow$	$start(J) \leqslant end(I)$
I not-meets J	$\Leftrightarrow$	$end(I) \neq start(J)$
I not-overlaps J	$\Leftrightarrow$	$(start(J) \leqslant start(I)) \vee (end(I) \leqslant start(J)) \vee (end(J) \leqslant end(I))$
I not-starts J	$\Leftrightarrow$	$(start(I) \neq start(J)) \vee (end(J) \leqslant end(I))$
I not-during J	$\Leftrightarrow$	$(start(I) \leqslant start(J)) \vee (end(J) \leqslant end(I))$
I not-finishes J	$\Leftrightarrow$	$(end(I) \neq end(J)) \vee (start(I) \leqslant start(J))$
I not-equal J	$\Leftrightarrow$	$(start(I) \neq start(J)) \vee (end(I) \neq end(J))$
I not-after J	$\Leftrightarrow$	$start(I) \leqslant end(I)$
I not-met-by J	$\Leftrightarrow$	$end(J) \neq start(I)$
I not-overlapped-by J	$\Leftrightarrow$	$(start(I) \leqslant start(J)) \vee (end(J) \leqslant start(I)) \vee (end(I) \leqslant end(J))$
I not-started-by J	$\Leftrightarrow$	$(start(J) \neq start(I)) \vee (end(I) \leqslant end(J))$
I not-contains J	$\Leftrightarrow$	$(start(J) \leqslant start(I)) \vee (end(I) \leqslant end(J))$
I not-finished-by J	$\Leftrightarrow$	$(end(J) \neq end(I)) \vee (start(J) \leqslant start(I))$

Fig. 12. The PA-disjunctions translating the negation of the thirteen basic interval relations.

4. Managing disjunctions

The expressive power of a *TL*-graph can be significantly increased by augmenting it with a set of disjunctions of PA-relations (PA-disjunctions). In particular, for binary PA-disjunctions a substantially larger class of interval relations than SIA can be translated into point-relations. This can be seen by observing that given an interval relation R expressed as a subset S of the set A of the thirteen basic interval relations (see Fig. 1), R holds for a pair of intervals if and only if none of the relations in $A - S$ hold; i.e., asserting that R holds is equivalent to asserting that the conjunction of negations of the relations in $A - S$ holds. The translation into point-relations of the negation of each basic interval relation is given in Fig. 12, where $start(I)$ and $end(I)$ indicate the starting and end points of the interval I (analogously for the interval J), and it is assumed that $start(I) < end(I)$ (analogously for J). Since there are only two basic relations (*overlaps* and *overlapped-by*) whose negations correspond to ternary PA-disjunctions (all the other disjunctions are at most binary), the number of relations in IA that can be translated into a collection of binary PA-disjunctions is *at least* 2^{11} (2048) against the 188 of SIA. In fact, all the interval relations corresponding to the set $\{\{overlaps, overlapped\text{-}by\} \cup M\}$, where M is one of the 2^{11} subsets of $A - \{overlaps, overlapped\text{-}by\}$, are translatable into a collection of binary PA-disjunctions. This is just a lower bound because among the remaining set N of $2^{13} - 2^{11}$ interval relations there are other translatable relations.

For example, if R is the interval relation corresponding to the set $S = \{equal, starts, started\text{-}by, finishes, finished\text{-}by, during, contains, after, overlapped\text{-}by, met\text{-}by\}$, then $A - S$ is $\{before, meets, overlaps\}$ and the point-relations translating *not-before*, *not-meets* and *not-overlaps* are respectively:

$$start(J) \leqslant end(I), \tag{i}$$

$$start(J) \neq end(I), \tag{ii}$$

$$(start(J) \leqslant start(I)) \vee (end(I) \leqslant start(J)) \vee (end(J) \leqslant end(I)). \tag{iii}$$

From (i) and (ii) we can derive

I {*before,after*} J	$\Leftrightarrow$	$(end(I) < start(J)) \vee (end(J) < start(I))$
I {*before,met-by*} J	$\Leftrightarrow$	$(end(I) < start(J)) \vee (end(J) = start(I))$
I {*meets,after*} J	$\Leftrightarrow$	$(end(I) = start(J)) \vee (end(J) < start(I))$
I {*meets,met-by*} J	$\Leftrightarrow$	$(end(I) = start(J)) \vee (end(J) = start(I))$
I {*before,after,meets*} J	$\Leftrightarrow$	$(end(I) \leqslant start(J)) \vee (end(J) < start(I))$
I {*before,after,met-by*} J	$\Leftrightarrow$	$(end(I) < start(J)) \vee (end(J) \leqslant start(I))$
I {*before,meets,met-by*} J	$\Leftrightarrow$	$(end(I) \leqslant start(J)) \vee (end(J) = start(I))$
I {*meets,after,met-by*} J	$\Leftrightarrow$	$(end(I) = start(J)) \vee (end(J) \leqslant start(I))$
I {*before,after,meets,met-by*} J	$\Leftrightarrow$	$(end(I) \leqslant start(J)) \vee (end(J) \leqslant start(I))$

Fig. 13. Translation of the interval disjointness relations into PA-disjunctions.

$$start(J) < end(I), \tag{iv}$$

and from (iii) and (iv) we obtain

$$(start(J) \leqslant start(I)) \vee (end(J) \leqslant end(I)). \tag{v}$$

So, we have proved that R can be translated into (iv) and (v), and hence by binary PA-disjunctions.

It is interesting to note that among the translatable relations in M there are the disjointness relations (see Fig. 13). Moreover, PA-disjunctions also allow the representation of many non-binary interval relations such as (I *meets* J) *or* (K *before* H) (I ,J, K, H intervals) which are not in the Interval Algebra.

In this section we present efficient algorithms for determining the consistency of a timegraph augmented by a set of PA-disjunctions which we call a *disjunctive timegraph.*

Definition 4.1. A *disjunctive timegraph* ($\mathcal{D}$*-timegraph*) is a pair $\langle T, D\rangle$ where T is a timegraph and D is a set of binary PA-disjunctions involving only point variables in T.

The notion of a consistent *TL*-graph is generalized to a $\mathcal{D}$-timegraph $\langle T, D\rangle$ by requiring that for each PA-disjunction in D it is possible to select one of the disjuncts in such a way that the resulting collection of selected PA-relations can be consistently added to T. We call this set of selected disjuncts an *instantiation* of D in T, and the task of finding such a set *deciding* D relative to T.

Once we have an instantiation of D, we can easily solve the problem of finding a consistent scenario by adding the instantiation to T and using a topological sort algorithm [8, 38]. Moreover, the task of checking whether a relation R between two time points x and y is entailed by a $\mathcal{D}$-timegraph $\langle T, D\rangle$ can be reduced to the problem of finding an instantiation of D in an augmented version of T. We add the relation $x\overline{R}y$ to T (where $\overline{R}$ is the negation of R), obtaining a new timegraph T', and then check if (a) T' is consistent, and (b) D can be decided relative to the explicit graph of T' (if any). The original $\mathcal{D}$-timegraph entails xRy just in case one of (a), (b) does not hold.

In general, in order to decide a set of binary disjunctions we can perform a search in the set of the 2^m possible ways of choosing the disjuncts (for m disjunctions). This search is necessarily exponential in the worst case (assuming P $\neq$ NP) since, as we proved in [13], the problem of determining the consistency of a $\mathcal{D}$-timegraph is NP-complete even when the allowed disjunctions are limited to "not between" relations (i.e.

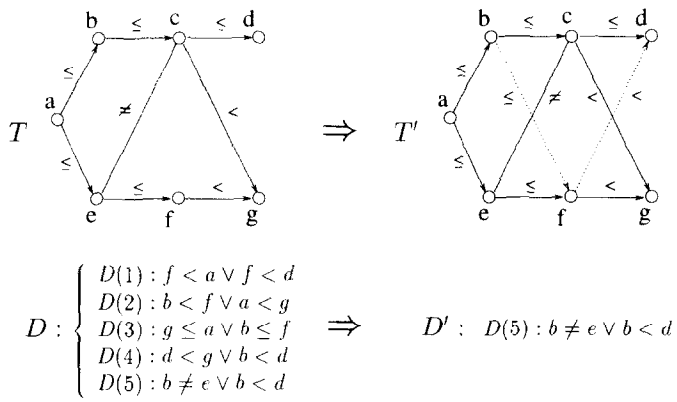


Fig. 14. A $\mathcal{D}$-timegraph $\langle T, D\rangle$ and the corresponding $\mathcal{D}$-timegraph $\langle T', D'\rangle$ obtained by applying the pruning rules to the disjunctions in D using T.

3-point relations of the form $xR_1w \vee zR_2x$, where $R_1, R_2 \in \{<, \leqslant\}$ and $w < z$) and the timegraph contains only $<$ relations.

Given a disjunctive timegraph $\langle T, D\rangle$, the algorithm we have developed for deciding D relative to T consists of two main steps:

(1) Prune the search space by reducing D to a subset D' of D and producing a timegraph T' such that D has an instantiation in T if and only if D' has an instantiation in T'.

(2) Search for an instantiation of D' in T' by using backtracking.

We first describe some powerful pruning rules on which the first step is based, and then present an efficient search algorithm which combines a form of *selective backtracking* [7,33] with *chronological backtracking*. In Section 5.2 the efficiency of the proposed techniques is evaluated experimentally.

4.1. Preprocessing

The set of disjunctions of a $\mathcal{D}$-timegraph $\langle T, D\rangle$ can be reduced to a significantly smaller subset by applying some *pruning rules* to each disjunction $D(i) = xR_1y \vee wR_2z$ in D (x,y,w, and z time points, R_1, R_2 PA-relations, and $i = 1..m$). These rules detect cases where the timegraph T already entails the disjunction (allowing its removal), or entails the negation of a disjunct (leaving only an ordinary PA-relation). For example, consider the $\mathcal{D}$-timegraph $H = \langle T, D\rangle$ of Fig. 14. H can be transformed into an equivalent $\mathcal{D}$-timegraph $H' = \langle T', D'\rangle$ where T' differs from T by having two additional edges (indicated by dotted arrows) and D' consists of only one of the disjunctions in D. In fact:

- $D(2)$ is redundant because T entails $a < g$;
- $D(4)$ can be eliminated because it is tautologically true (if we assume the negation of the first disjunct, i.e., $\neg(d < g)$, i.e., $g \leqslant d$, then we have $c < d$ from the graph and hence the second disjunct, $b < d$, is true);

- neither $f < a$ nor $g \leqslant a$ can be consistently added to T and hence the second disjuncts of $D(1)$ $(f < d)$ and of $D(3)$ $(b \leqslant f)$ must take part in any instantiation of D.

More formally, we can define three rules for eliminating a disjunction $D(i) = xR_1y \vee wR_2z$ from D, possibly producing a set Q of PA-relations which must take part in any instantiation of D:

(1) *T-derivability.* If $T \vdash xR_1y$ or $T \vdash wR_2z$, then $D := D - \{D(i)\}$.

(2) *T-tautology.*
  (i) If $T \cup \{x\overline{R}_1y\} \vdash wR_2z$ then $D := D - \{D(i)\}$.
  (ii) If $T \cup \{w\overline{R}_2z\} \vdash xR_1y$ then $D := D - \{D(i)\}$.

(3) *T-resolution.*
  (i) If $T \cup \{xR_1y\}$ is consistent and $T \cup \{wR_2z\}$ is inconsistent, then $D := D - \{D(i)\}$ and $Q := Q \cup \{xR_1y\}$.
  (ii) If $T \cup \{wR_2z\}$ is consistent and $T \cup \{xR_1y\}$ is inconsistent, then $D := D - \{D(i)\}$ and $Q := Q \cup \{wR_2z\}$.
  (iii) If both $T \cup \{xR_1y\}$ and $T \cup \{wR_2z\}$ are inconsistent, then the $\mathcal{D}$-timegraph $\langle T, D\rangle$ is inconsistent

where $T \vdash xR_iy$ $(i \in \{1,2\})$ holds if and only if xR_iy can be derived from T using Theorem 2.11, $T \cup \{xR_1y\}$ is the timegraph obtained by adding to T the graphical representation of xR_1y (analogously for $T \cup \{wR_2z\}$), $\overline{R}_1$ is the negation of R (analogously for $\overline{R}_2$), Q is initially empty.

With respect to the example of Fig. 14, $D(2)$ can be eliminated from D by the application of rule (1), $D(4)$ by rule (2) and $D(1)$, $D(3)$ by rule (3). (It is worth noting that if these inferences are made in sequence, and "resolvents" are *immediately* added to T, then we can also eliminate $D(5)$. However, we do not currently exploit this fact, since adding to T requires updating to maintain the timegraph data structures—e.g., nextgreaters—and ensure explicitness, and with our current algorithms the cost of repeated updates can outweigh the gains.)

Rule (1) is a special case of rule (2). We keep them separated because the time required for applying rule (2) may be too great; rule (1) just requires application of the timegraph query algorithms, whereas rule (2) calls for a (temporary) addition to the timegraph. In fact, when one of the disjuncts of the disjunction is an "=" or "<" relation, the addition to the timegraph of the negation of the disjunct can determine the creation of new $\neq$-diamonds that have to be reduced, and of new $\leqslant$-cycles that have to be collapsed before verifying the entailment of the second disjunct. In these cases a rule weaker than rule (2) may be preferred. In particular, for disjunctions d of the form $x < y \vee w < z$ we can use the following rule:

(2′) *Restricted T-tautology.* If $T \vdash (x < z \wedge w \leqslant y) \vee T \vdash (x \leqslant z \wedge w < y)$, then $D := D - \{d\}$.

This rule is applied to a disjunction only if it cannot be removed by rule (1), that is the timegraph does not contain $<$-paths from x to y and from w to z (otherwise the disjunction would be eliminated by rule (1)). In fact, under this condition, if T entails $x < z$ and $w \leqslant y$, or $x \leqslant z$ and $w < y$, and we assume the negation of the first disjunct $(y \leqslant x)$ or of the second disjunct $(w \leqslant z)$, the resulting timegraph entails $x < y$ or $w < z$ respectively (see Fig. 15).

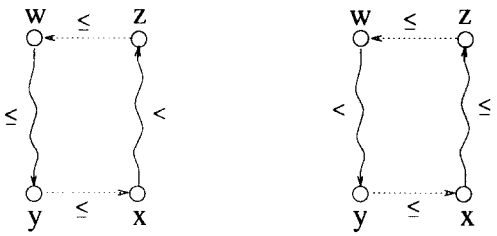


Fig. 15. Two cases in which the precondition of restricted *TL*-tautology is satisfied. Dotted edges represent the assumed relations. "Wavy" arcs are $\leqslant$-paths or $<$-paths.

Rule (2′) is sound but not complete in the sense that there can be some disjunctions that the rule does not eliminate but which the full T-tautology can remove. However, for timegraphs, applying rule (1) and restricted T-tautology is more efficient then applying rule (2) since checking the preconditions of rule (1) and rule (2′) can be accomplished just by using query algorithms. The choice between using the full T-tautology and the combination of rule (1) and restricted T-tautology (or other weaker rules) depends on how much effort we want to dedicate to the preprocessing step and how much to the search step.[6]

If rule (3(iii)) can be applied to a disjunction d in D, then there is no instantiation of D because the addition of either disjunct of d makes the resulting timegraph inconsistent. When only one of (3(i)) and (3(ii)) can be applied to d, the disjunct of d which can be consistently added to the graph is called the *T-resolvent* of d.[7]

If D contains a disjunct of the form $s < t$ (or $s \leqslant t$), and T has a $\leqslant$-path ($<$-path) from the vertex corresponding to t to the vertex corresponding to s, we say that the $\mathcal{D}$-timegraph contains a *$<$-quasicycle* (quasicycle hereafter) determined by the disjunct $s < y$ ($s \leqslant t$). For example, the $\mathcal{D}$-timegraph of Fig. 14 contains a quasicycle determined by $g \leqslant a$ (the first disjunct of $D(3)$). When rule (3) is applied to a disjunction having a disjunct which determines a quasicycle we call this *quasicycle elimination*. The application of rule (3) is an efficient operation especially when the disjuncts are $<$-relations. In fact, in these cases quasicycle elimination can be applied just by checking the existence of a $\leqslant$-path in the metagraph of the timegraph. For example, for eliminating $x < y \vee w < z$ it suffices to verify that the timegraph entails $z \leqslant w$ (rule (3(i))) or that it entails $y \leqslant x$ (rule (3(ii))).

The result of applying the pruning rules to all the disjunctions of D is a subset D' of D and a set Q of T-resolvents. The PA-relations of Q can then be added to the original graph producing a new timegraph T' which is used by the search step to find an instantiation of D'. In fact, as the following theorem asserts, the original problem of deciding D in T is equivalent to the problem of deciding D' in T'.

[6] The worst-case behavior of rule (2) can be improved by reformulating it so that *both* $x\overline{R}_1 y$ and $w\overline{R}_2 z$ are added to T, and then the consistency of T is checked. This would avoid the potential $O(e^2)$ cost of $\neq$-diamond reduction in favor of $O(e)$ consistency checking. However, it is still advantageous to use (2′) whenever possible.

[7] This terminology reflects the strong similarity of T-resolution to a particular form of "Theory Resolution" proposed by Stickel [35].

Theorem 4.2. *Given a $\mathcal{D}$-timegraph $\langle T, D\rangle$, D has an instantiation in T if and only if $D - \{d\}$ has an instantiation in T', where d is any disjunction in D which can be eliminated by the application of any pruning rule, $T' = T$ if d is eliminated either by T-derivability, T-tautology or by restricted T-tautology, and $T' = T \cup \{vRw\}$ if d is eliminated by T-resolution and vRw is the T-resolvent of d.*

Proof (Sketch). The proof follows from Theorem 2.11 and from the fact that each disjunction is binary. □

Various strategies are possible for preprocessing the set of disjunctions using the pruning rules. Here we have adopted the simplest one in which the rules are applied to each disjunction once and the set of T-resolvents generated is added to the timegraph at the end of the process. This strategy is very efficient since it is based on query algorithms and does not require updating to maintain the timegraph data structures. Moreover, as will be shown in Section 5.2, this technique is particularly effective when the timegraph is not sparse.

A more complete strategy, though more computationally expensive, is to add the T-resolvents to the graph as soon as they are produced and to iterate the application of the rules till no further disjunction can be eliminated. This strategy is still polynomial but since determining the consistency of a disjunctive timegraph is NP-complete, in general it does not guarantee that when the iteration stops without having detected an inconsistency, the graph is indeed consistent. However, we show that the above strategy is complete for an important class of PA-disjunctions translating the interval relations in the ORD-Horn subclass of IA. The proofs of the next claims are based on the following definitions and facts from [29]:

(1) Disjunctions of PA-relations of the form $a = b$, $a \leqslant b$, $a \neq b$ are called ORD clauses. ORD clauses containing at most one literal (PA-relation) of the form $a = b$ or $a \leqslant b$ and any number of literals of the form $a \neq b$ are called ORD-Horn clauses [29].

(2) The theory *ORD* that axiomatizes "=" as an equivalence relation and "$\leqslant$" as a partial ordering over the equivalence classes is a Horn theory [29].

(3) A finite set Ω of ORD clauses has a model in the real numbers (is R-satisfiable) *iff* $\Omega \cup ORD_\Omega$ is satisfiable, where ORD_Ω denotes the axioms of *ORD* instantiated to all the point-variables mentioned in Ω [29].

Proposition 4.3. *The set of the PA-relations entailed by a timegraph can be translated into a logically equivalent set of ORD-Horn clauses.*

Proof. The proof trivially follows from the fact that each PA-relation can be reformulated in terms of "=" and "$\leqslant$" and their negation. □

Proposition 4.4. *A $\mathcal{D}$-timegraph $\langle T, D\rangle$ is consistent* iff *$R \cup D \cup ORD_R$ is satisfiable, where R is the set of PA-relations entailed by T, and D is a set of ORD-Horn clauses.*

Proof. The proof trivially follows from Property (3) and Proposition 4.3, and from the fact that $ORD_{R \cup D} = ORD_R$. □

Theorem 4.5. *There exists a polynomial strategy for applying the pruning rules which is complete for determining the consistency of a $\mathcal{D}$-timegraph $\langle T, D \rangle$, where D is a set of binary ORD-Horn clauses.*

Proof. Suppose that the pruning rules are applied by using a strategy such that the T-resolvents are added to the timegraph as soon as they are produced, and the application of the rules is iterated till no further disjunctions can be eliminated. Since each iteration takes polynomial time and the maximum number of iterations corresponds to the number of disjunctions in D, this strategy is polynomial.

It is clear that if the preprocessing using this strategy terminates by reporting inconsistency then the initial $\mathcal{D}$-timegraph is not consistent. The key point is to show that termination without detection of inconsistency entails consistency of $\langle T, D \rangle$.

Let $\langle T', D' \rangle$ be the $\mathcal{D}$-timegraph at termination, and R' the set of PA-relations entailed by T'. Since positive unit resolution is known to be refutation-complete for Horn theories [20], by Property (2) and Propositions 4.3 and 4.4 it suffices to show that unit resolution applied to $R \cup D \cup ORD_R$, or equivalently to $R' \cup D' \cup ORD_{R'}$, cannot derive any unit clauses (PA-relations) that are not in R'. We know that no unit resolution is possible between any PA-relation in R' and any disjunction in D' (otherwise, the disjunct resolved against would have been detected not to be consistent with the timegraph during preprocessing). So, the only unit resolutions that may be possible are between PA-relations in R' and clauses in $ORD_{R'}$. But this will not give any new PA-relation that we cannot already "read off" T', i.e., a unit clause that does not belong to R'. So, there are no unit resolutions of any kind that lead to any PA-relation (unit clause) that we do not already "know" (that does not belong to R').

Hence, we can conclude that if the preprocessing does not stop by reporting inconsistency, no contradiction is derivable, i.e., the initial disjunctive timegraph is consistent. □

Given a set I of interval relations in the ORD-Horn subclass that can be translated into a set H of ORD-Horn clauses, we write T_I for the timegraph built from the set of the unary clauses of H, and D_I for the set of the remaining (binary) clauses of H.

Theorem 4.6. *The consistency of a set I of interval relations in the ORD-Horn subclass can be polynomially decided by determining the consistency of T_I and by using the preprocessing step of the algorithm for deciding D_I relative to the timegraph built from T_I.*

Proof (Sketch). The proof follows from Theorem 4.5 and the fact that the clauses in D_I are at most binary [29]. By Theorem 3.2 consistency checking of the set of unary clauses of H is accomplished in polynomial time during the construction of T_I. If T_I is consistent then Theorem 4.5 guarantees that the preprocessing step of the algorithm for deciding D_I relative to T_I is sufficient for determining the consistency of I. □

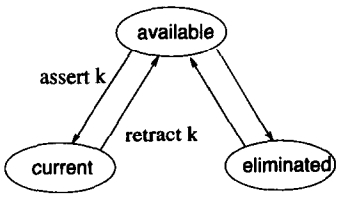


Fig. 16. Possible transitions for the status of a disjunct k.

4.2. *Partially selective backtracking*

Once the initial set of disjunctions has been reduced by the application of the pruning rules, an arbitrary total order is imposed on the remaining disjunctions, and the search for an instantiation of them is activated. In this section we describe an algorithm for binary disjunctions of inequalities. The algorithm can easily be generalized to disjunctions containing $\neq$ relations, while for $\leqslant$ and $=$ relations some further effort is generally needed because we have to deal with new $\leqslant$-cycles and implicit $<$ relations that the addition of $<$ and $\leqslant$ relations to the timegraph can produce.

We first introduce some terminology, part of which is borrowed from [7], and the general backtracking strategy.

4.2.1. *Terminology and backtracking strategy*

A disjunction is indicated with $D(j)$ $(1 \leqslant j \leqslant m$, for m disjunctions), where j corresponds to the position of the disjunction in the ordered set. The two disjuncts of a disjunction $D(j)$ are denoted by $d(j,1)$ and $d(j,2)$.

Each disjunct has a status associated with it that can be *available, current* or *eliminated.* A disjunct is available if it hasn't been tried yet. Initially all the disjuncts are available. Fig. 16 shows the possible transitions for the status of a disjunct. An available disjunct becomes current when it is selected as part of the current (attempted) instantiation of D. At this point the corresponding edge is added to the timegraph, and the disjunction to which it belongs becomes *decided.* A disjunct changes status from available to eliminated when the addition of the corresponding edge to the timegraph would make the resulting graph inconsistent. During backtracking a disjunct can change status from current to available and from eliminated to available. When it changes from current to available the corresponding edge is retracted.

In general, after each addition and retraction the timegraph should be restored to explicit form. However this is not always necessary if all the disjuncts are PA-relations in $\{<, \neq\}$. In fact in these cases any addition or retraction in the timegraph does not induce new $\leqslant$-cycles that are not $<$-cycles, and even though new $\neq$ relations can induce new implicit $<$-relations, checking if a disjunction of the form $x < y$ determines a quasicycle does not require the graph to be explicit. Moreover, for disjuncts of the form $x \neq y$, an inconsistency can arise only when x,y are alternative names of the same vertex.

We indicate with T^j the timegraph resulting after deciding $D(j)$. The set of current disjuncts taking part in a quasicycle determined by an eliminated disjunct $d(i,j)$ is a

set of *antagonists* of $D(i)$. Note that a disjunct can determine more than one quasicycle and hence the corresponding disjunction can have more than one set of antagonists. However, we will always consider only one of them: the set of current disjuncts taking part in the $\leqslant$-path identified by the procedure which checks for the existence of the quasicycle. The *culprit* of an eliminated disjunct $d(i,j)$ (written as *culprit*(i,j)) is the *most recently* decided disjunction responsible for its elimination; i.e., it is the most recently decided disjunction $D(h)$ $(1 \leqslant h < i)$ such that:

(1) one of the disjuncts of $D(h)$ is available;

(2) the other disjunct of $D(h)$ is among the antagonists of $D(i)$.

In order to simplify the explanation of the method, we assume that all the disjuncts are different PA-relations. This assumption can easily be relaxed by checking whenever a disjunction is examined if one of its available disjuncts is equal to the current disjunct of a disjunction already decided, and modifying the relevant data structures accordingly.

The search for an instantiation is conducted by deciding each disjunction in turn, adding the chosen disjunct to the graph, until all disjunctions are decided or an impasse is reached, i.e., the next disjunction $D(i)$ cannot be consistently decided either way (i.e. without adding quasicycles). In the latter case, we backtrack to the culprit of one of the disjuncts of $D(i)$. When both the culprits are null the search proceeds by backtracking chronologically. This is illustrated in the following example.

4.2.2. An example

Let L be a $\mathcal{D}$-timegraph $\langle T, D\rangle$ where T is the timegraph of Fig. 17 and D is the following set of disjunctions:

$$D(1) : b < f \vee n < g,$$
$$D(2) : f < m \vee n < f,$$
$$D(3) : c < g \vee m < b,$$
$$D(4) : f < b \vee h < e,$$
$$D(5) : n < f \vee e < g.$$

The graph T_1 shows the state of the disjunctions when the search has reached $D(5)$. Dotted edges correspond to chosen (current) disjuncts. The label on a dotted edge indicates the disjunction to which the disjunct belongs. When the label for a chosen disjunct of a disjunction $D(i)$ has a number in brackets, this means that the other disjunct of $D(i)$ has already been eliminated, and the number indicates the culprit of the examined disjunct.

For example, the label D4[1] on the edge from h to e indicates that $D(1)$ is the culprit of $f < b$, the first disjunct of $D(4)$ (when $D(4)$ was decided, the addition of $f < b$ to the graph would have created the cycle b, f, b). Once we have decided $D(4)$, an impasse is reached because $D(5)$ cannot be decided. So, we have to select a decided disjunction for backtracking. Since neither $D(3)$ nor $D(4)$ satisfy conditions 1 and 2, disjunctions $D(1)$ and $D(2)$ are considered because $D(1)$ is the culprit of the eliminated disjunct of $D(4)$, and $D(2)$ is the culprit of $n < f$, the second disjunct of $D(5)$. Redeciding $D(1)$ using $n < g$ will break the cycle b, f, b, while redeciding

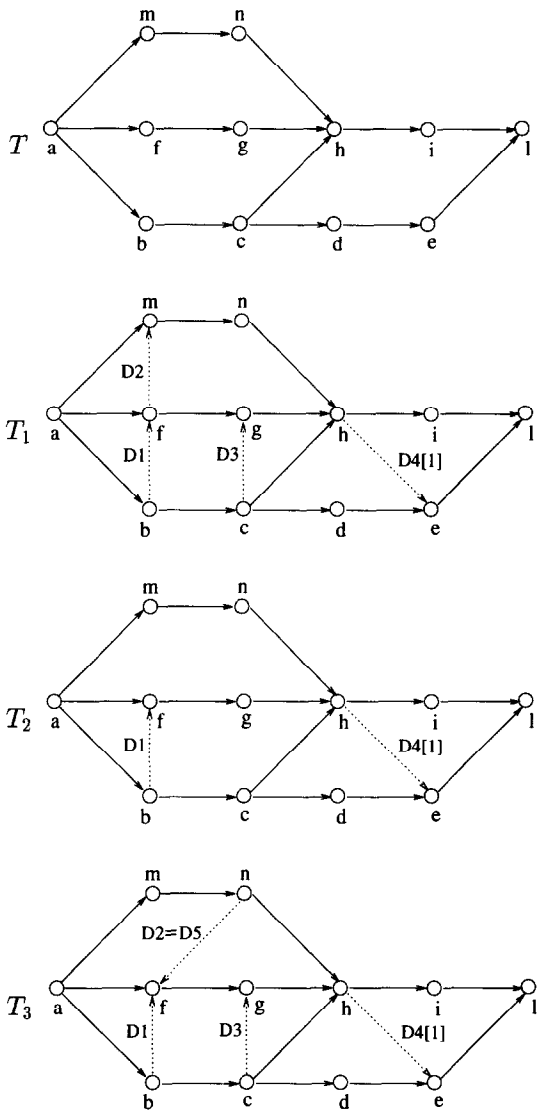


Fig. 17. Example of the search for an instantiation.

$D(2)$ using $n < f$ will break the cycle f, m, n, f. $D(2)$ is selected because it has been decided more recently than $D(1)$.

The graph T_2 of Fig. 17 shows the state of the search after backtracking. Note that in general the decision for $D(3)$ needs to be reconsidered, whereas we do not have to redecide $D(4)$ since its chosen disjunct involves no new cycles. (Moreover, its culprit remains the same since the other way of deciding $D(4)$ ($f < b$) still creates a quasicycle).

Finally, T_3 shows the instantiation found by the search.

```
ALGORITHM: DECIDE-DISJUNCTIONS
INPUT: a timegraph T and a set D of preprocessed disjunctions
OUTPUT: the vector DSJ if <T,D> is consistent, nil otherwise

1.  i:= 1; fail:= false; done:=false; FOR i = 1 TO m CULPRIT[i]:=nil;
2.  WHILE (fail = false) and (done = false) DO
3.   IF DSJ[i] = 0 THEN {both the disjuncts are available}
4.     IF d(i,1) does not determine a quasicycle THEN
5.        mark d(i,1) current, DSJ[i]:= 1, and CULPRIT[i]:= i
6.     ELSE
7.       BEGIN CULPRIT[i]:= culprit(i,1);
8.       IF d(i,2) does not determine a quasicycle THEN
9.          mark d(i,2) current and DSJ[i]:= 2
10.      ELSE BEGIN
11.           IF culprit(i,2) > CULPRIT[i] THEN CULPRIT[i]:=culprit(i,2);
12.           IF CULPRIT[i] = nil THEN
13.             BEGIN {chronological backtracking}
14.             q:= the highest j such that j < i and DSJ[i] = 1,
                    nil if such a j does not exist;
15.             IF q is not nil THEN
                  BEGIN {restore disjunctions between D(q) and D(i)}
16.               CULPRIT[q]:= nil; mark d(q,1) eliminated;
17.               FOR t = q + 1 TO i
18.                 IF CULPRIT[t] >= q THEN mark the disjuncts of D(t)
                       available, DSJ[t]:= 0, and CULPRIT[t]:= nil;
19.               i:= q {backtracking to D(q)}
                  END
20.             ELSE fail:= true {the D-timegraph is inconsistent}
                END
21.           ELSE {selective backtracking}
22.                q:= CULPRIT[i] and run through steps 16. to 19.
              END
        END
23.  ELSE BEGIN {DSJ[i] = 1, d(i,1) eliminated and d(i,2) available}
24.       CULPRIT[i]:= i;
25.       IF d(i,2) does not determine a quasicycle THEN
26.          mark d(i,2) current, DSJ[i]:= 2
27.       ELSE BEGIN CULPRIT[i]:= culprit(i,2);
28.            IF CULPRIT[i] = nil THEN run through steps 14. to 20.
29.            ELSE q:= CULPRIT[i] and run through steps 16. to 19.
               END
          END
30.  IF NOT fail THEN IF all disjunctions are decided THEN i:=m+1
                      ELSE i:=lowest j such that j>=i and D(j) is undecided;
31   IF i > m THEN done:= true
    END{WHILE};
32. IF fail = true THEN RETURN nil ELSE RETURN the vector DSJ.
```

Fig. 18. Algorithm for deciding a set of m preprocessed disjunctions.

4.2.3. Basic algorithm and data structures

The backtracking algorithm of Fig. 18 decides (when it is possible) the disjunctions following the arbitrary order imposed earlier. The algorithm uses two data structures:

DSJ and CULPRIT.

DSJ is a vector of dimension m in which the status of the disjuncts of the disjunctions is maintained. DSJ$[j]$ is an integer in $\{0,1,2\}$ with the following meaning:

- DSJ$[j] = 0$ if $D(j)$ is not decided and both the disjuncts are available;
- DSJ$[j] = 1$ if $D(j)$ is decided and $d(j,1)$ is the current disjunct;
- DSJ$[j] = 2$ if $D(j)$ is decided and $d(j,2)$ is the current disjunct.

During the process of deciding a disjunction $D(j)$, when both the disjuncts are available, the disjunct $d(j,1)$ is always tried before the disjunct $d(j,2)$. As a consequence of this, when $d(j,2)$ is current $d(j,1)$ is always eliminated.

CULPRIT is a vector of dimension m which is used to store for each disjunction the CULPRIT of the eliminated disjunct (if any). If $D(j)$ is a decided disjunction without an eliminated disjunct, then CULPRIT$[j] = j$; if $D(j)$ is not yet decided, then CULPRIT$[j]$ = nil.

A disjunction $D(j)$ cannot be decided when both its disjuncts take part in a quasicycle determined in T^{j-1}. In this case there can be two culprits for $D(j)$, and the one corresponding to the most recently decided disjunction is stored in CULPRIT[j] and can be used to select a disjunction to backtrack to. A simple chronological backtracking mechanism would always choose as backtrack point the most recently decided disjunction $D(k)$ with an available disjunct, and all the disjunctions $D(h)$ between $D(k)$ and $D(j)$ would be restored (i.e. DSJ$[h] = 0$ and CULPRIT$[h]$ = nil). But exploiting our data structures, if $D(j)$ has an antagonist which is not null (i.e. CULPRIT$[j] \neq$ nil), we can use it to jump back directly to the most recently decided disjunction whose current disjunct takes part in a quasicycle determined by the disjunct of $D(j)$. Moreover, not all the disjunctions between $D(k)$ and $D(j)$ need to be restored, in fact all the disjunctions $D(h)$ such that CULPRIT$[h] < k$ can be left unchanged. The reason for this is that DSJ$[h]$ can have a value different from the current one (at the moment of backtracking) only if there is a backtrack to a disjunction which is equal to or which precedes CULPRIT$[h]$. In fact, if $k >$ CULPRIT$[h]$ then the eliminated disjunct of $D(h)$ will continue to determine a quasicycle in each timegraph T^i $(k \leqslant i < h)$, no matter how $D(k)$ and the further disjunctions preceding $D(h)$ are decided after the backtrack to $D(k)$.

This form of selective backtracking can significantly prune the search in comparison with ordinary chronological backtracking. However, for each j, CULPRIT$[j]$ indicates only one disjunction which is responsible for an impasse reached during the search. As discussed in [7,33], to obtain a complete selective backtracking algorithm we could augment the information stored in the CULPRIT data structure, maintaining for each eliminated disjunct *all* the previously decided disjunctions which are responsible for its elimination (i.e. its set of antagonists) instead of just the latest one. However, the computational space required by the algorithm performing full selective backtracking would then be $O(m^2)$, where m is the number of disjunctions. This bound would be unacceptable for large graphs if the number of disjunctions is comparable to the number n of time points or greater (it can be as high as n^4).

So, in order to guarantee completeness while retaining linear space complexity, whenever an undecidable disjunction $D(j)$ with an empty antagonist is reached, the algorithm performs a chronological backtrack to the first preceding disjunction $D(k)$ $(k < j)$ with

an available disjunct, and each disjunction between $D(h)$ and $D(k)$ lacking an antagonist preceding k is restored. If such a disjunction does not exist, then there is no instantiation of the original set of disjunctions.

4.2.4. Forward propagation

When the initial timegraph is very sparse, the dearth of constraints imposed by the graph can protract the search for an instantiation of the disjunctions and the algorithm may perform an unacceptable number of backtracks (see Section 5.2). In this section we propose a technique which in practice can dramatically reduce the number of backtracks in such cases.

Forward propagation of a decided disjunction $D(i)$ consists of checking all disjunctions $D(j)$ not yet decided and following $D(i)$ in the ordering (i.e. such that $i < j \leqslant m$), to determine whether at least one of the disjuncts of $D(j)$ can be consistently added to T^h $(i < h < j)$, where h is the index of the most recently decided disjunction at the moment of checking $D(j)$. If only one disjunct of $D(j)$ determines an inconsistency, then its status is set to eliminated and the other disjunct is made current. If both disjuncts determine a quasicycle then $D(j)$ cannot be decided and we perform a jump back to CULPRIT$[j]$ (if CULPRIT$[j] \neq$ nil) or a chronological backtrack to the first disjunction preceding $D(j)$ with an available disjunct (if CULPRIT$[j]$ = nil).

Fig. 19 shows the algorithm for achieving the forward propagation of a decided disjunction. It uses an additional data structure called D-SET which is a vector of dimension m where D-SET$[i]$ is the set I of disjunction indices such that $k \in I$ if $D(k)$ is a disjunction which has been decided by the forward propagation of $D(i)$. In Appendix B the search algorithm enhanced by the inclusion of the forward propagation is reported. Note that for backtracks performed by this new algorithm, we restore disjunctions in accordance with the basic algorithm, and in addition for each disjunction $D(i)$ that is updated we also restore the set of decided disjunctions depending on it (i.e. the disjunctions indicated by D-SET$[i]$).

The space complexity overhead introduced by D-SET is negligible because there can never be more disjunction indices stored in all the locations of the vector than the number of disjunctions. Furthermore, since DSJ and CULPRIT are also vectors of dimension m, it follows that the space complexity of the whole algorithm is $O(n + e + m)$, where n is the number of point variables, e the number of PA-relations and m the number of disjunctions.

5. Experimental results

The algorithms described in the previous sections have been implemented in a temporal reasoning system called *TimeGraph II* (TG-II).[8] In this section we report some results from large scale tests we have conducted on a SUN SPARCstation 10.

The experiments consist of two main classes. The first is aimed at testing the performance and the scalability of building and of querying a timegraph, which are polynomial

[8] TG-II is written in Common Lisp and it is available by inquiry to the authors.

```
ALGORITHM: PROPAGATE
INPUT: a decided disjunction D(i) {i:1..m}
OUTPUT: nil if an inconsistency is detected; m+1 if all the disjunctions
        are decided; a disjunction index j such that DSJ[j]=0  otherwise

1.  FOR p = i + 1 TO m
2.    IF DSJ[p]=0 THEN
3.      IF only one disjunct d(p,k) of D(p) does not create a
           quasicycle THEN mark d(p,k) current, CULPRIT[p]:=
           culprit(p,3-k), DSJ[p]:= k, and add p to D-SET[i]
4.      ELSE
5.        IF both d(p,1) and d(p,2) determine a quasicycle THEN
            BEGIN
6.          IF culprit(p,1) = culprit(p,2) = nil THEN
              BEGIN {chronological backtracking}
7.            q:= the highest j such that j < i and DSJ[j]=1,
                  or nil if such a j does not exist;
8.            IF q is not nil THEN
                BEGIN {restore disjunctions between D(q) and D(i)}
9.              CULPRIT[q]:= nil; mark d(q,1) eliminated;
10.             FOR t = q to i
11.               IF CULPRIT[t] >= q THEN
                    BEGIN
12.                 IF t is not equal to q THEN mark the disjuncts
                      of D(t) available, DSJ[t]:=0, CULPRIT[t]:=nil;
13.                 FOR EACH h in D-SET[t] mark d(h,1) and d(h,2)
                        available, DSJ[h]:=0, CULPRIT[h]:=nil,
                        and D-SET[h]:=nil
                    END
14.             i:= q {backtrack to D(q)}
                END
15.           ELSE RETURN nil {the D-timegraph is inconsistent}
              END
16.         ELSE
              BEGIN {selective backtracking}
17.           CULPRIT[p]:= max(culprit(p,1),culprit(p,2));
18.           q:= CULPRIT[p]; run through steps 9. to 14.
              END
19.         RETURN i
            END
20.   RETURN the lowest j such that j > i and DSJ[j] = 0, or
            m + 1 if such a j does not exist.
```

Fig. 19. Algorithm for the forward propagation of a decided disjunction.

tasks. In these experiments we were mostly interested in data sets which tend to fall into chains, since our algorithms are designed to do especially well in such cases compared with traditional constraint propagation algorithms. Moreover, we believe that this assumption about the temporal structure of the information is typically satisfied in story comprehension problems [28] and in many planning domains such as TRAINS [1].

The second class of experiments concerns the algorithms for deciding the consistency of a disjunctive timegraph. Since this task is NP-hard these experiments were aimed not

Table 2
Statistics for the construction of a timegraph

points	CPU-time	chains	length-chain	max-chain
100	74	20.3	6.10	48.9
200	191	40.5	8.18	97.5
300	308	54.6	12.21	154.1
400	470	73.3	12.28	206.2
500	592	96.5	11.47	244.7
1000	1711	182.8	14.36	508.8

only at testing the scalability of the method proposed, but also at exploring the space of the problems without any strongly biased assumptions about the structure of the information, to enable us to identify interesting parameters in terms of which effective heuristics can be formulated.

5.1. Construction and querying of a timegraph

Table 2 reports statistics about building timegraphs for randomly generated consistent sets of relations. For each number n of time points considered, the test procedure generates n sets of relations, each containing a number of relations equal to $n \cdot INT(\log_2 n)$ and constraining different pair of variables. For example, with $n = 500$, the procedure generates 500 timegraphs from 500 data sets each of which contains 4000 relations. The table reports the average CPU-time (milliseconds) for building the timegraph, the average number of chains, the average length of the chains and the average maximum length of the chains.

Given a set S of n time points with indices $1, \ldots, n$, the first m points (S')—with m a random number between 1 and n—are chosen to represent distinct elements whose time order is the same as the order of their indices; the remaining $n - m$ points (S'') are randomly assigned to coincide with points in S'. Relations are generated randomly by choosing a pair i, j ($i < j$) of indices. Specifically, if both i and j are among the m distinct points, then the relation iRj is generated, where R is randomly taken from $\{<, \leqslant, \neq\}$, with the percentage of $\neq$ relations kept low (2%). If i or j (or both) is one of the $n - m$ points in S'', the corresponding point in S' is considered. For example, if $n = 100$, $m = 70$, $i = 50$, $j = 80$ and j was assigned to 50, then a relation between i and j is randomly taken from $\{\leqslant, \geqslant, =\}$. Since we were mostly interested in measuring TG-II's performance with data sets likely to allow chain formation, the pairs of points were generated using a geometric distribution with expected value 3.

Fig. 20 compares the CPU-times for reducing $\neq$-diamonds in a timegraph and in the corresponding network of relations (van Beek's algorithm [37]). For each value marked on the curves, 500 randomly generated data sets of $\leqslant$ relations were considered. Five *irreducible* $\neq$ relations (i.e. that do not induce any implicit $<$ relation of Fig. 3(a)) were then added to each data set. The high efficiency of *NCD-NCA* compared to van Beek's algorithm derives mainly from the fact that the number of $\neq$-diamonds examined by *NCD-NCA* was nearly constant for all the data sets generated. Note however that this experiment is more concerned with comparing the scalability of the two approaches

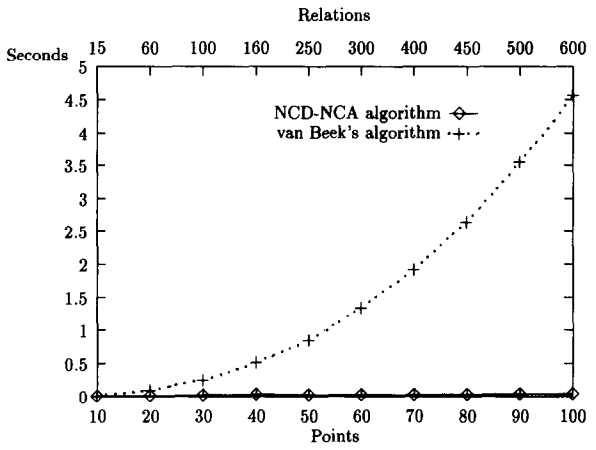


Fig. 20. CPU-time for reducing $\neq$-diamonds.

than with directly comparing the relative CPU-times.[9] In this sense the results of the experiment show that our algorithm for dealing with $\neq$ relations is much more efficient than van Beek's algorithm.

Other experiments show that querying a timegraph is on average a fast operation. For example, the average CPU-time over 100,000 queries on 100 randomly generated timegraphs with 500 points and 4000 relations was 2.8 milliseconds.

Finally, we should mention some recent experiments conducted by Yampratoom and Allen [43] comparing the performance of Timegraph I and II with several temporal reasoning systems based on constraint propagation algorithms—TimeLogic [23], MATS [22], Tachyon [6] and TMM [9, 10]—in which the timegraph approach proved by far the most efficient for large data sets generated for the TRAINS world [1]. (TMM came closest to matching TG-II's performance, but does not handle $\neq$ or provide completeness guarantees even for the Convex Point Algebra. On the other hand, like TimeGraph I it handles metric information.)

5.2. *Deciding binary PA-disjunctions*

Tables 3 and 4 report the number of backtracks (the mean, the standard deviation and the maximum value) and the average CPU-time (milliseconds) required for deciding a set of 30 disjunctions of form $x < y \vee w < z$ using a timegraph built from a collection of *convex* PA-relations constraining 30 time points.[10] Table 3 pertains to the basic algorithm, while Table 4 pertains to the algorithm performing the forward propagation

[9] A direct comparison of the CPU-times cannot be adequate because the two algorithms have different outputs: a timegraph in *NCD-NCA*, a "minimal network" in van Beek's algorithm.

[10] The convex PA-relations are all the relations of PA except "$\neq$" [39,41]. The reason we haven't considered $\neq$ relations is that for the kind of disjunctions we were dealing with, the information provided by these relations is exploited neither by the preprocessing step nor by the search algorithms.

Table 3
DECIDE-DISJUNCTIONS: number of backtracks and average CPU-time for deciding 30 binary disjunctions of $<$-relations in databases of convex PA-relations constraining 30 points (14,000 problems)

Convex	Backtracks			Average
relations	Mean	Deviation	Max.	CPU-time
30	482.98	3686.99	92424	1160
50	122.17	2143.80	91516	351
75	11.69	311.46	13760	47
100	0.28	3.80	96	19
150	0.019	0.47	20	10
200	0	0	0	6
250	0	0	0	5

Table 4
DECIDE-DISJUNCTIONS-WITH-PROPAGATION: number of backtracks and average CPU-time for deciding 30 binary disjunctions of $<$-relations in databases of convex PA-relations constraining 30 points (14,000 problems)

Convex	Backtracks			Average
relations	Mean	Deviation	Max.	CPU-time
30	1.13	6.29	142	104
50	0.65	12.27	547	70
75	0.12	0.49	4	29
100	0.03	0.24	3	20
150	0.005	0.09	2	10
200	0	0	0	7
250	0	0	0	5

during the search (see Fig. 19 and Appendix B) which was applied to the same data sets used for Table 3.

For each number r of convex PA-relations considered, 2000 randomly generated $\mathcal{D}$-timegraphs were built, each of which was created in the following way: first, a set of convex PA-relations S was generated following a method similar to the one used for testing the construction of a timegraph (see Section 5.1) but using the uniform distribution instead of the geometric one in choosing the pair of points to be constrained;[11] second, a timegraph T was constructed from S; finally, a set D of disjunctions of the form $x < y \vee w < z$ was built by randomly generating each of them in such a way that:

- x, y, w and z are point variables taking part in at least one of the relations in S;
- $x \neq y$, $w \neq z$ and the pair (x, y) is different from the pair (w, z);
- neither $x < y$ nor $w < z$ are in S.

It is interesting to observe that the number of backtracks shown in Table 3 decreases dramatically when the number of PA-relations used to build the timegraph is greater than three times the number of the time points. The main reason for this is that in general

[11] By choosing the uniform distribution we have relaxed the assumption made in the previous section that the data sets generated are likely to allow chain formation.

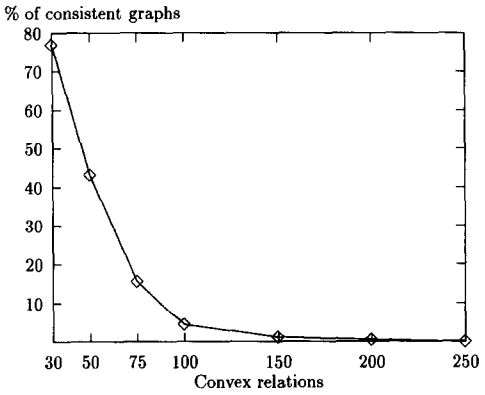


Fig. 21. Distribution of the consistent $\mathcal{D}$-timegraphs over 14,000 data sets with 30 binary disjunctions of $<$-relations and 30 time points.

the more constraints are imposed on the timegraph, the more disjunctions are eliminated by the pruning rules, and hence the easier the problem becomes for the backtracking step. For example, with 30 convex PA-relations constraining 30 points only 11% of the disjunctions were eliminated on average, while with 75 convex PA-relations constraining 30 points this percentage was 58.5%.

Fig. 21 shows the distribution of the consistent data sets with respect to the number of PA-relations forming the timegraph. This curve indicates that when the timegraph is not particularly sparse the probability of the $\mathcal{D}$-timegraph being consistent is much smaller that the probability of its being inconsistent. Since the percentage of consistent data sets generated goes down drastically when the timegraph is not sparse, we have repeated the test of Table 3 by generating only consistent data sets. [12] The results of this experiment are qualitatively identical to those obtained when consistency is not enforced, and so they show that when the timegraph is not particularly sparse our method performs efficiently regardless of whether the information provided is consistent or not.

When the timegraph is sparse, there are some cases in which the basic algorithm performs a large number of backtracks (see Table 3) and hence incurs large CPU time costs. Fortunately, as shown by the elevated values of the standard deviation when the number of convex PA-relations is between 30 and 75, these computationally expensive cases are relatively rare. This has been further confirmed experimentally by computing their percentage over 30,000 $\mathcal{D}$-timegraphs with 30 time points, 20 convex PA-relations and 30 disjunctions. Table 5 shows that the number of cases for which more than 400 backtracks were performed is limited to 0.9% of all $\mathcal{D}$-timegraph generated, while the percentage of $\mathcal{D}$-timegraphs requiring at most 10 backtracks was 96.01%.

Table 4 shows that when the timegraph is sparse, the use of the forward propagation technique during the search can dramatically reduce the number of backtracks. However, the CPU-time we need for propagating decided disjunctions can be too high for larger data sets. The main reason for this is that quasicycle elimination requires $O(\hat{e})$ time

[12] This was obtained by adding to the data set generator the extra requirement that for each disjunction at least one of its disjuncts is consistent.

Table 5
Backtracking requirements for "sparse" data sets.[a]

Backtracks	% of $\mathcal{D}$-timegraphs
|0..10|	96.01
(10..50|	1.8
(50..100|	0.53
(100..200|	0.43
(200..300|	0.21
(300..400|	0.12
> 400	0.9

[a] Distribution of 30,000 $\mathcal{D}$-timegraphs with respect to the number of backtracks performed by the basic search algorithm (30 disjunctions, 20 PA-relations and 30 time points).

(for $\hat{e}$ meta-edges in the timegraph).[13] It follows that a general good heuristic for large data sets is to prefer the use of forward propagation when the initial timegraph is sparse or when the number of disjunctions to be decided is particularly high with respect to the number of convex PA-relations, and to use the basic algorithm in other cases.

The curves in Fig. 22 show the results of other experiments aimed at testing the scalability of the proposed approach. In this experiment we have considered only consistent data sets following the method described above. The two curves of the first graph show the average CPU-time (seconds) required for deciding sets of disjunctions of size n when there are n time points with $8n$ and $2n \log n$ simple PA-relations (with the logarithm truncated to its integer part). The numbers attached to the points on the curve indicate the percentage of the disjunctions eliminated by preprocessing. While in the case of $8n$ relations this percentage decreases when n increases, for $2n \log n$ relations it tends to be constant (wavering between 88.8 and 91.3).[14]

The curve in the second graph shows the average CPU-time required by more sparse graphs ($4n$ PA-relations) for which forward propagation has been used during the search.[15]

Finally, the third graph shows that for the 16,000 problems considered our algorithms tend to perform polynomially (quadratic time for the curves of the first graph and cubic time for the curve of the second) since the previous curves approximate straight lines on a log–log scale.

6. Conclusions

In this paper we addressed the problem of scalability in temporal reasoning, proposing a collection of new algorithms that are implemented in a temporal reasoning system

[13] In fact, as noted in Section 4.1, checking the existence of a quasicycle determined by a disjunct of the form $r < s$ can be accomplished by querying whether the timegraph entails $s \leqslant r$, which in the worst case requires linear time in the number of meta-edges in the timegraph.

[14] In order to simplify the figure these numbers are not shown.

[15] These experiments were conducted on a SUN SPARCstation 2 and hence a comparison with the results in the previous graph is inappropriate.

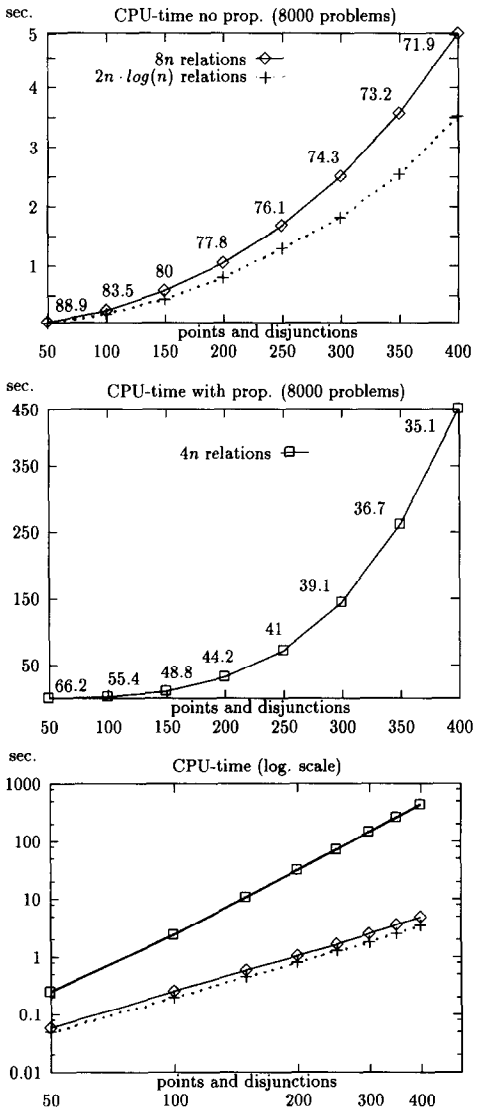


Fig. 22. Scalability of the algorithms for deciding a set of disjunctions.

called *TimeGraph* II (TG-II).

In the first part of the paper we dealt with relations in the Point Algebra. In comparison with the constraint propagation approach, in our system both space and time complexity are reduced significantly for many practical applications. In fact, the space complexity depends on the size of the set of the stipulated relations and on a limited number of $\neq$ relations (a subset of those forming $\neq$-diamonds), and not on the number of time points (as in the constraint propagation approach). Instead of computing the closure of the set of relations, we build a timegraph providing a collection of data structures

that allow efficient deduction of relations at query time. Experimental results show that building a timegraph is much faster than computing the minimal network and that, on average, querying relations is very efficient. The larger lesson here is that avoiding precomputation of implicit relations and exploiting the inherent temporal structure (in our case, the chain-like structure) of the information in certain important classes of domains can yield dramatic improvements in practical performance.

Though timegraphs were first proposed for efficient temporal reasoning in story understanding, there are reasons for regarding this approach as well-suited for planning applications. As we have noted, sets of temporal constraints that were generated in the context of the TRAINS project at Rochester (independently of our work) conformed well with our assumptions. In addition, van Beek [38] estimated some parameters for sets of IA-relations arising in the kind of constraint-based planning proposed by Allen and Koomen [5]. He found that about 25% of the pairs of intervals were constrained by interval relations, and the great majority of these relations were pointizable. These are just the sorts of conditions under which our consistency algorithm for $\mathcal{D}$-timegraphs is most efficient. Note that a consistent scenario (and hence a workable plan) is easily found by topological sorting once the disjunctions of a $\mathcal{D}$-timegraph have been consistently instantiated by our algorithm.

Current and future work concerns the design of efficient algorithms for adding new relations to the timegraph dynamically and the integration of metric relations involving deadlines, durations and absolute times from which derive information about the partial order of the time points maintained in the timegraph.[16]

In the second part of the paper we investigated a major extension of the Point Algebra to include binary disjunctions of PA-relations. This extension allows the representation of many useful relations outside PA including, in particular, disjointness between intervals of time. We addressed the problem of determining consistency by providing an efficient method which is based on two steps: preprocessing the initial set of disjunctions to reduce it to a logically equivalent subset; and searching for an instantiation of the remaining disjunctions, i.e. a set of disjuncts (one for each disjunction) which can be consistently added to the given timegraph.

The preprocessing step is worst-case polynomial and is based on some pruning rules which exploit the information stored in the timegraph to eliminate disjunctions. There are several polynomial strategies for applying these rules, including one which is complete for checking the consistency of interval relations in the ORD-Horn subclass of IA. The choice of strategy depends on how much effort one wants to dedicate to the preprocessing step and how much to the search step. Though in general consistency testing is NP-complete for disjunctive information, experimental results show our algorithms for instantiating disjunctions to be approximately polynomial in practice.

It is an interesting question whether the preprocessing step by itself can provide a good "approximate" method for determining consistency. Further research in this direction might be aimed at providing approximate consistency-checking algorithms for

[16] These were handled in the original implementation of timegraphs [28,32], but as noted $\neq$ was not handled and also $<$ and $\leqslant$ relations entailed via metric relations were not extracted in a deductively complete way.

disjunctive temporal information that operate more efficiently and accurately than current approximate techniques based on constraint propagation [2, 39].

Finally, we are examining the possibility of improving the disjunction instantiation algorithm by excluding disjunctions from the instantiation whenever either of their disjuncts can be consistently assumed independently of how any other disjunctions are instantiated (certain instances of this property are easily detectable). This variant is especially relevant to planning, where we may not want to impose an order on actions until there are compelling reasons to do so.

Acknowledgments

We wish to thank James Allen for drawing our attention to the problem of scalability and for many fruitful discussions. The work of the first author was carried out in part during a visit at the Computer Science Department of the University of Rochester (NY) with the support of the Italian National Research Council (CNR), and in part at IRST in the context of the MAIA project and the CNR projects "Sistemi Informatici e Calcolo Parallelo", and "Pianificazione Automatica". The second author was supported by Rome Lab Contract F30602-91-C-0010 and Boeing Contract W88104.

Appendix A

We first give an informal description of the algorithm for refining the nextgreater links for a chain c, and then we present its pseudocode.

The algorithm searches for $<$-paths that go from one cross-connected vertex of c to another, starting with an outgoing cross-edge and ending with an incoming cross-edge. One search is potentially done for each cross-connected vertex with an outgoing cross-edge, starting at the vertex with the highest possible pseudotime and working "backward". Each search records (or updates) two pieces of information at each vertex v it visits, `maxless(v)` and `maxleq(v)`. This information is used to avoid duplication of effort in path-following. (As discussed below, a search can often be terminated at a previously visited vertex). Vertices where maxless and maxleq information is recorded are saved, and the values are reset to their defaults (`nil`) at the end. Their meanings are:

- `maxless(v)` is the maximum pseudotime of any vertex on c which is known to be a $<$-ancestor of v (is `nil` if there is none, interpreted as $-\infty$);
- `maxleq(v)` is the maximum pseudotime greater than `maxless(v)` of any vertex on c which is known to be a $\leqslant$-ancestor of v (is `nil` if there is none).

Intuitively, if `maxless(v)` for some arbitrary vertex v is the pseudotime of vertex u on chain c, this tells us that if we can find a $\leqslant$-path from v to some vertex w on c, we will have found a $<$-path from u to w (and so we can make `nextgreater(u)` point no further than w). Similarly if `maxleq(v)` is the pseudotime of vertex u on chain c, this tells us that if we can find a $<$-path from v to some vertex w on c, we will have found a $<$-path from u to w.

Note that since we will search from cross-connected vertices with higher pseudotimes

first, when we find a $<$-path to a vertex v to which we previously found a $<$-path, we can terminate that search path, since the previous path started at a higher pseudotime (so if some paths from v go back to chain c, we already found them and made the strongest possible updates to nextgreater pointers based on them). Similarly, when we find a $\leqslant$-path to a vertex v to which we previously found a $<$-path or $\leqslant$-path, we do not need to search further from v.

Once we have found all paths from a vertex *vstart* on chain c to other cross-connected vertices on c, and updated `nextgreater(vstart)` accordingly, we can move backward link-by-link from *vstart* and update each of their nextgreater pointers, until we get to the another cross-connected vertex with outgoing cross-edges. Then we initiate another search. (However, the search from *vstart* can be omitted if `nextgreater(vstart)` already points to a vertex preceding `nextin(vstart)` on the chain. In that case `nextgreater(vstart)` could not possibly be tightened by an "external" $<$-path.)

The search from a vertex *vstart* is conducted by means of an *open* list of vertices still to be expanded (searched from). One complication is that vertex expansion needs to take account not only of successors via by cross-edges, but also successors corresponding to *nextout* links on chains that have been entered via cross-edges.

Note that we assume that the following parameters are available for each chain c: `firstout(c)`, `lastout(c)`, `firstin(c)`, `lastin(c)`, where `firstout(c)` is the first (lowest-pseudotime) cross-connected vertex with one or more outgoing cross-edges, `lastout(c)` is the last (highest-pseudotime) cross-connected vertex with one or more outgoing cross-edges, etc. Also, the first vertex of a chain (whether or not it is a cross-connected vertex) is assumed to be given by `firstvertex(c)`.

```
ALGORITHM: REFINE-NEXTGREATERS-LINKS
INPUT: A time chain c
OUTPUT: The nextgreater links for c

1.  vstart := lastout(c);
2.  IF vstart=nil {no outgoing cross-edges} THEN return;
3.  {Find the highest-pseudotime vertex from which a search may
     succeed; this vertex must have a non-nil nextin pointer}
4.  WHILE nextin(vstart)=nil DO
5.    IF prevout(vstart)=nil
6.       THEN RETURN ELSE vstart := prevout(vstart);
7.  rmax:= rank(lastin(c)) {We can terminate search paths when
                            their rank gets higher than that of the
                            latest relevant chain re-entry point}
8.  visited := nil {list of vertices, for later restoration of
                    maxless, maxleq}
9.  REPEAT {search from successive cross-connected vertices}
      {Revise max. rank to which to search}
10.   rmax := min(rank(nextgreater(vstart)),rmax);
11.   t := pseudotime(vstart);
12.   OPEN := (list vstart);
13.   newval := nil; {the search from vstart has not yet updated
                      nextgreater(vstart)}
14.   WHILE OPEN is not empty (nil) DO
15.    v := pop(OPEN);
```

```
16.    IF chain(v)= c AND v not equal to vstart
          {chain c has been re-entered}
17.       THEN IF (nextgreater(vstart)=nil OR
                  pseudotime(v) < pseudotime(nextgreater(vstart)))
                  AND maxless(v) is not nil
                  THEN BEGIN
18.                    nextgreater(vstart) := v;
19.                    newval := v
                       END
20.       ELSE BEGIN
21.            W:={(w,l) | w is a cross-edge successor of v with
                   rank(w) < rmax, where the edge label on that edge
                   is l, i.e., < or <=};
22.            IF chain(v) is not equal to c AND nextout(v) is not nil
                         AND rank(nextout(v)) < rmax
23.               THEN W := W union {(nextout(v),l)}, where l = ''<''
                       if pseudotime(nextgreater(v)) <= pseudotime
                       (nextout(v)), and l = ''<='' otherwise};
24.             FOR each (w,l) in W DO
25.                IF l = ''<'' OR maxless(v) is not nil {<-path to w}
26.                  THEN IF maxless(w)=nil OR maxleq(w)=t
                             THEN BEGIN
27.                               maxless(w) := t;
28.                               IF maxleq(w)=t THEN maxleq(w):=nil
                                  END
29.                  ELSE {nonstrict path to w}
30.                     IF maxleq(w)=nil AND maxless(w)=nil
31.                        THEN maxleq(w) := t;
32.                put w on OPEN and on visited without duplication
                   END {FOR}
             END {ELSE}
      END {WHILE};
      {If the search from vstart led to an update of nextgreater,
      then chain-ancestors of vstart need to be updated as well,
      back to the next point (if any) where another search can begin}
33.   IF newval is not nil THEN
         BEGIN
34.      u := vstart;
35.      If prevout(vstart)=nil THEN
36.         umin:= firstvertex(c) ELSE umin:=prevout(vstart);
37.      WHILE u is not equal to umin DO
38.         u:= predecessor of u in chain c (i.e., with next-lower
                pseudotime);
39.         IF nextgreater(u)=nil OR
               pseudotime(newval) < pseudotime(nextgreater(u))
40.            THEN nextgreater(u) := newval
41.            ELSE u := umin {terminates update loop}
            END {WHILE}
         END {IF};
42. vstart := prevout(vstart)
43. UNTIL vstart = nil;
    {Restore maxless and maxleq values to nil}
44. FOR each v in visited DO maxless(v) := nil; maxleq(v):= nil.
```

Appendix B

```
ALGORITHM: DECIDE-DISJUNCTIONS-WITH-PROPAGATION
INPUT: a timegraph T and a set D of preprocessed disjunctions
OUTPUT: the vector DSJ if <T,D> is consistent, nil otherwise

1. FOR i = 1 to m + 1 CULPRIT[i]:= nil;
2. i:= 1; fail:= false; done:= false;
3. WHILE (done = false) AND (fail = false) DO
4.   IF DSJ[i] = 0 THEN {both the disjuncts are available}
5.     IF d(i,1) does not determine a quasicycle THEN mark d(i,1)
6.       current, DSJ[i]:= 1, CULPRIT[i]:= i, i:= PROPAGATE(D(i))
8.     ELSE
9.       BEGIN CULPRIT[i]:= culprit(i,1);
10.      IF d(i,2) does not determine a quasicycle THEN
11.        mark d(i,2) current, DSJ[i]:= 2, i:= PROPAGATE(D(i))
12.      ELSE
13.       BEGIN
14.       IF culprit(i,2) > CULPRIT[i] THEN CULPRIT[i]:=culprit(i,2);
15.       IF CULPRIT[i] = nil THEN
            BEGIN {chronological backtracking}
16.         q:= the highest j such that j < i and DSJ[j]=1,
                or nil if such a j does not exist;
17.         IF q is not nil THEN
              BEGIN {restore disjunctions between D(q) and D(i)}
18.           CULPRIT[q]:= nil; mark d(q,1) eliminated;
19.           FOR t = q to i
20.             IF CULPRIT[t] >= q THEN BEGIN
21.               IF t is not equal to q THEN mark d(t,1) and d(t,2)
                    available, DSJ[t]:= nil, CULPRIT[t]:= nil;
22.               FOR EACH h in D-SET[t] mark d(h,1) and d(h,2)
                  available, DSJ[h]:=0, CULPRIT[h]:=nil, D-SET[h]:=nil;
                  END
23.           i:= q {backtracking to D(q)}
              END
24.         ELSE fail:= true {the D-timegraph is inconsistent}
            END
25.       ELSE {selective backtracking}
26.            q:= CULPRIT[i] and run through steps 18. to 23.
          END
        END
27.   ELSE BEGIN CULPRIT[i]:= i;
28.       IF D(i,2) does not determine a quasicycle THEN
29.         mark d(i,2) current, DSJ[i]:= 2, i:= PROPAGATE(D(i));
30.        ELSE
31.          BEGIN CULPRIT[i]:= culprit(1,2);
32.          IF CULPRIT[i] = nil THEN run through steps 16. to 24.
33.          ELSE  q:= CULPRIT[i] and run through steps 18. to 23.
             END
          END;
34. IF i > m THEN done:= true ELSE IF i = nil THEN fail:= true
    END{WHILE}
35. IF fail=true THEN RETURN nil ELSE RETURN the vector DSJ.
```

References

[1] J.F. Allen and L. Schubert, The TRAINS project, Tech. Report 91-1, Department of Computer Science, University of Rochester, Rochester, NY (1991).

[2] J.F. Allen, Maintaining knowledge about temporal intervals, *Commun. ACM* **26** (1) (1983) 832–843.

[3] J.F. Allen, Towards a general theory of action and time, *Artif. Intell.* **23** (2) (1984) 123–154.

[4] J.F. Allen, Temporal reasoning and planning, in: J.F. Allen, H. Kautz, R. Pelavin and J. Tenenberg, eds., *Reasoning about plans* (Morgan Kaufmann, San Mateo, CA, 1991).

[5] J.F. Allen and J.A. Koomen, Planning using a temporal world model, in: *Proceedings IJCAI-83*, Karlsruhe, Germany (1983) 741–747.

[6] R. Arthur and J. Stillman, Temporal reasoning for planning and scheduling. User's guide: Preliminary release, Tech. Report, Artificial Intelligence Laboratory, General Electric Research and Development Center (1992).

[7] M. Bruynooghe, Solving combinatorial search problems by intelligent backtracking, *Inf. Process. Lett.* **12** (1) (1981).

[8] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms* (MIT Press, Cambridge, MA, 1990).

[9] T. Dean, Using temporal hierarchies to efficiently maintain large temporal databases, *J. ACM* **36** (4) (1989) 687–718.

[10] T. Dean and D.V. McDermott, Temporal data base management, *Artif. Intell.* **32** (1987) 1–55.

[11] R. Dechter, I. Meiri and J. Pearl, Temporal constraint networks, *Artif. Intell.* **49** (1991) 61–95.

[12] J. Dorn, Temporal reasoning in sequence graphs, in: *Proceedings AAAI-92*, San Jose, CA (1992) 735–740.

[13] A. Gerevini and L. Schubert, On point-based temporal disjointness (Research Note), *Artif. Intell.* **70** (1–2) (1994) 347–361.

[14] A. Gerevini and L. Schubert, Efficient temporal reasoning through timegraphs, in: *Proceedings IJCAI-93*, Chambéry, France (1993) 648–654.

[15] A. Gerevini and L. Schubert, An efficient method for managing disjunctions in qualitative temporal reasoning, in: *Proceedings Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR94)*, San Francisco, CA (1994).

[16] A. Gerevini and L. Schubert, On computing the minimal labels in time point algebra networks, *Comput. Intell.*, to appear.

[17] A. Gerevini, L. Schubert and S. Schaeffer, Temporal reasoning in TimeGraph I-II, *SIGART Bull.* **4** (3) (1993) 21–25.

[18] M. Ghallab and A. Mounir Alaoui, Managing efficiently temporal relations through indexed spanning trees, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 1297–1303.

[19] M. Golumbic and R. Shamir, Algorithms and complexity for reasoning about time, in: *Proceedings AAAI-92*, San Jose, CA (1992) 741–746.

[20] L. Henschen and L. Wos, Unit refutations and Horn sets, *J. ACM* **21** (1974) 590–605.

[21] H. Kautz, A formal theory of plan recognition and its implementation, in: J.F. Allen, H. Kautz, R. Pelavin and J. Tenenberg, eds., *Reasoning about Plans* (Morgan Kaufmann, San Mateo, CA, 1991).

[22] H. Kautz and P. Ladkin, Integrating metric and qualitative temporal reasoning, in: *Proceedings AAAI-91*, Anaheim, CA (1991).

[23] J.A.G.M. Koomen, The timelogic temporal reasoning system, Tech. Report 231, Computer Science Department, University of Rochester, Rochester, NY (1988).

[24] P. Ladkin and R. Maddux, On binary constraint networks, Tech. Report KES.U.88.8, Kestrel Institute, Palo Alto, CA (1988).

[25] P. Ladkin and R. Maddux, On binary constraint problems, *J. ACM* **41** (3) (1994) 435–469.

[26] P. Ladkin and A. Reinefeld, Effective solution of qualitative interval constraint problems, *Artif. Intell.* **57** (1) (1992) 105–124.

[27] I. Meiri, Combining qualitative and quantitative constraints in temporal reasoning, in: *Proceedings AAAI-91*, Anaheim, CA (1991).

[28] S.A. Miller and L.K. Schubert, Time revisited, *Comput. Intell.* **6** (1990) 108–118.

[29] B. Nebel and H.J. Bürckert, Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra, Research Report RR-93-11, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany (1993).

[30] K. Nökel, *Temporally Distributed Symptoms in Technical Diagnosis*, Lecture Notes in Computer Science **517** (Springer, Berlin, 1991).

[31] L.K. Schubert, M.A. Papalaskaris and J. Taugher, Determining type, part, colour, and time relationships, *Computer* **16** (1983) 53–60.

[32] L.K. Schubert, M.A. Papalaskaris and J. Taugher, Accelerating deductive inference: special methods for taxonomies, colours, and times, in: N. Cercone and G. McCalla, eds., *The Knowledge Frontier: Essays in the Representation of Knowledge* (Springer, Berlin, 1987) 187–220.

[33] M. Shanahan and R. Southwick, *Search, Inference and Dependencies in Artificial Intelligence* (Ellis Horwood, Chichester, UK, 1989).

[34] F. Song and R. Cohen, The interpretation of temporal relations in a narrative, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 745–750.

[35] M.E. Stickel, Automated deduction by theory resolution, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 1181–1186.

[36] R. Tarjan, Depth first search and linear graph algorithms, *SIAM J. Comput.* **1** (2) (1972) 215–225.

[37] P. van Beek, Reasoning about qualitative temporal information, in: *Proceedings AAAI-90*, Boston, MA (1990) 728–734.

[38] P. van Beek, Reasoning about qualitative temporal information, *Artificial Intelligence*, 58(1-3):297–321, (1992).

[39] P. van Beek and R. Cohen, Exact and approximate reasoning about temporal relations, *Comput. Intell.* **6** (1990) 132–144.

[40] M. Vilain and H. Kautz, Constraint propagation algorithms for temporal reasoning, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 377–382.

[41] M. Vilain, H. Kautz and P. van Beek, Constraint propagation algorithms for temporal reasoning: a revised report, in: *Readings in Qualitative Reasoning about Physical Systems* (Morgan Kaufmann, San Mateo, CA, 1990) 373–381.

[42] R. Weida and D. Litman, Terminological reasoning with constraint networks and an application to plan recognition, in: B. Nebel, W. Swartout and C. Rich, eds., *Proceedings Third International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, (1992) 282–293.

[43] E. Yampratoom and J. Allen, Performance of temporal reasoning systems, *SIGART Bull.* **4** (3) (1993).